UNIVERSITY OF CALIFORNIA,
IRVINE

A Semantic Approach To Data Management for Smart Spaces

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Peeyush Gupta

Dissertation Committee:
Professor Sharad Mehrotra, Chair
Professor Michael J. Carey
Professor Chen Li
Professor Nalini Venkatasubramanian

2022

DEDICATION

To Baba Saheb - My grandfather.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost I am extremely grateful to my advisor, Professor Sharad Mehrotra for his invaluable advice, continuous support, and patience during my PhD. His immense knowledge and experience have encouraged me in my academic research and daily life. I am glad that I had the opportunity to work with him.

I would like to thank all the members of my doctoral committee, Professor Michael Carey, Professor Chen Li, and Professor Nalini Venkatasubramanian, for providing me with valuable feedback on my thesis.

I would also like to thank Dhrubajyoti Ghosh, Shantanu Sharma and Roberto Yus for their invaluable suggestions. Technical discussions with them helped me immensely in improving my research and technical skills.

I am very thankful to Yiming, Sriram, Nada, Rahul, Mowhebat, Guangxue and all the other members of the TippersDB project. It would be impossible to develop the TippersDB system without their help. I am also grateful to Sameera, Primal, Joyce, Guoxi and other members of the ISG group who made my time at UCI more rewarding and fun.

Finally, I would like to express my gratitude to my wife Hiteshi. Without her tremendous understanding and encouragement, it would be impossible for me to complete my PhD. Also, I would like to thank my parents for supporting me during my PhD. I am forever indebted for their love and support.

The work reported in this thesis was supported by DARPA under Agreement No. FA8750-16-2-0021, and NSF Grants No. 1952247, 2133391, 2032525, 200899 and 2008993.

VITA

Peeyush Gupta

EDUCATION

Doctor of Philosophy in Computer Science **2022**
University of California, Irvine *Irvine, CA*

M.Tech in Computer Science **2013**
IIT Bombay *Mumbai, India*

B.Tech in Computer Science **2011**
The LNMIIT *Jaipur, India*

RESEARCH EXPERIENCE

Graduate Research Assistant **2019–2022**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2016–2019**
University of California, Irvine *Irvine, California*

REFEREED JOURNAL PUBLICATIONS

A Case for Enrichment in Data Management Systems **2022**
Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, and Shantanu Sharma

SIGMOD Record

Quest: Privacy-Preserving Monitoring of Network Data: A System for Organizational Response to Pandemics **2022**
Shantanu Sharma, Sharad Mehrotra, Nisha Panwar, Nalini Venkatasubramanian, Peeyush Gupta, Shanshan Han, and Guoxi Wang

IEEE TSC

Privacy-Preserving Secret Shared Computations using MapReduce **2021**
Shlomi Dolev, Peeyush Gupta, Yin Li, Sharad Mehrotra, Shantanu Sharma

IEEE TDSC

PANDA: Partitioned Data Security on Outsourced Sensitive and Non-sensitive Data **2020**
Sharad Mehrotra, Shantanu Sharma, Jeffrey D. Ullman, Dhrubajyoti Ghosh and Peeyush Gupta

ACM TMIS

REFEREED CONFERENCE PUBLICATIONS

TippersDB: A Sensor Observable Data Model For Smart Spaces **2022**
Peeyush Gupta, Sharad Mehrotra, Shantanu Sharma, Roberto Yus and Nalini Venkatasubramanian

ACM CIKM [Applied Research Track]

Supporting Complex Query Time Enrichment For Analytics **2023**
Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, and Shantanu Sharma

EDBT

JENNER: Just-in-time Enrichment in Query Processing **2022**
Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, Roberto Yus and Yasser Altowim

PVLDB

SmartSPEC: Customizable Smart Space Datasets via Event-Driven Simulations **2022**

Andrew Chio, Daokun Jiang, Peeyush Gupta, Georgios Bouloukakis, Roberto Yus, Sharad Mehrotra and Nalini Venkatasubramanian

IEEE PerCom

Prism: Private Verifiable Set Computation over Multi-Owner Outsourced Databases **2021**

Yin Li, Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, Nisha Panwar, and Shantanu Sharma

ACM SIGMOD

Concealer: SGX-based Secure, Volume Hiding, and Verifiable Processing of Spatial Time-Series Datasets **2021**

Peeyush Gupta, Sharad Mehrotra, Shantanu Sharma, Nalini Venkatasubramanian and Guoxi Wang

EDBT

A Privacy-Enabled Platform for COVID-19 Applications **2020**

Michael August, Christopher Davison, Mamadou Diallo, Dhrubajyoti Ghosh, Peeyush Gupta, Christopher Graves, Shanshan Han, Michael Holstrom, Pramod Khargonekar, Megan Kline, Sharad Mehrotra, Shantanu Sharma, Nalini Venkatasubramanian, Guoxi Wang and Roberto Yus

ACM SenSys Workshop

IoT Expunge: Implementing Verifiable Retention of IoT Data **2020**

Nisha Panwar, Shantanu Sharma, Peeyush Gupta, Dhrubajyoti Ghosh, Sharad Mehrotra and Nalini Venkatasubramanian

ACM CODASPY

SmartBench: A Benchmark For Data Management In Smart Spaces **2020**

Peeyush Gupta, Michael Carey, Sharad Mehrotra, Roberto Yus

PVLDB

OBSCURE: Information-Theoretic Oblivious and Verifiable Aggregation Queries **2019**

Peeyush Gupta, Yin Li, Sharad Mehrotra, Nisha Panwar, Shantanu Sharma, Sumaya Almanee

PVLDB

SOFTWARE

TippersDB `http://github.com/ucisharadlab/tdb`

TippersDB is a middleware system designed to build sensor-based smart space applications. TippersDB provides semantic abstraction over sensor data to application developers and optimizes translation of sensor data to semantic concepts.

SmartBench `http://github.com/ucisharadlab/benchmark`

SmartBench is a benchmark developed to test different database systems in their suitability for smart space applications. SmartBench is derived for a real smart space deployment and focuses on workloads resulting from (near) real-time applications as well as longer-term analysis of IoT data.

EnrichDB `http://github.com/DBrepo/enrichdb`

EnrichDB is a a system developed for just-in-time data enrichment. EnrichDB enriches data progressively in epochs and reduces wait time for the users by providing them with early results.

ABSTRACT OF THE DISSERTATION

A Semantic Approach To Data Management for Smart Spaces

By

Peeyush Gupta

Doctor of Philosophy in Computer Science

University of California, Irvine, 2022

Professor Sharad Mehrotra, Chair

Unprecedented growth in sensing, data capture devices, communication, and computing technologies have enabled us to capture and analyze almost every aspect of our lives. Such a prospect has resulted in the development of a myriad of applications in the cyber-physical systems, and the Internet of Things domain. While limitless possibilities exist, their realization has led to the grand challenge of designing a new class of systems that can provide seamless support for efficient and easier to program sensor-based applications. Furthermore, as sensor data processing often consists of complex compiled code, expensive machine learning, and signal processing code, systems that can scale to data generated by a myriad of interconnected sensors and devices embedded in the environment are needed.

This thesis presents TippersDB, a middleware system designed to build sensor-based smart space analytical applications. The intended goal of TippersDB is to ease the task of developing complex smart space applications, to develop an extensible system that allows diverse/heterogeneous sensors (with possibly overlapping functionalities) to co-exist and to be seamlessly integrated into the system without requiring application redesign, and to exploit the semantics of the physical world, as well as, sensor capabilities to optimize the applications. TippersDB provides a powerful data model that decouples semantic data about the application domain from sensor data using which the semantic data is derived. By support-

ing mechanisms to translate data, concepts, and queries between the two levels, TippersDB relieves the application developers from having to know or reason about either the type or location of sensors or write sensor specific code. In addition, it allows for multiple optimizations based on smart space semantics to improve query processing.

In particular, in this thesis, first we introduce the SmartBench benchmark that we developed to analyze the existing database technologies in terms of their strength/weaknesses and suitability in supporting IoT applications. Next, we introduce the TippersDB data model and describe its query-driven translation of sensor data. We then provide a summary of the system implementation and show a performance evaluation of TippersDB using IoT benchmark queries. This is followed by the description of the progressive query processing techniques used in TippersDB to reduce wait time for the users by providing them early results. Finally, we highlight the benefits of TippersDB through a case study.

Chapter 1

Introduction

Today, the information technology revolution, ongoing now for several decades, is focused on the next breakthrough in the form of ubiquitous deployment and adoption of IoT and smart space technologies. Such technologies create a distinct possibility to embed sensors in the physical world and continuously capture and analyze (in real-time) data from the sensors to create a digital representation of our lives – whether it be personal experiences, social interactions, or our interactions with engineered, cyber, or physical systems. Such a prospect has led to a myriad of directions of exploration, in academia and industry alike, in the form of cyber-physical systems, pervasive computing systems, and the Internet of things (IoT) systems. In domains such as emergency response, in-situ and mobile sensors coupled with participatory sensing can create awareness about the extent of damage, the affected population, and their emerging needs, leading to improved resource planning and improved response, thereby saving lives and property. Likewise, in domains such as healthcare and wellness, wearable devices, smart implants, and instrumented facilities/homes could lead to an awareness that provides deeper insight about ones' health, disease prevention, improved diagnosis, and medical practices.

With limitless possibilities, there has been a recent surge in efforts to refactor existing data management technologies and/or design new products to meet the requirements of these emerging applications. Today, multiple data management products advertise themselves as platforms for IoT applications [20]. These include standard relational systems [118], key-value stores, document databases [9], or specialized systems (such as time series stores that represent time-varying data and queries involving temporal operators and aggregation), scalable data ingest systems (such as Kafka [2]), stream processing systems (e.g., Storm [127], Spark Streaming [137], Flink [48]), and big data frameworks (such as Hadoop/Hive) that can support complex analytical pipelines on very large datasets.

While existing systems individually (or collectively) meet several data management requirements of IoT applications, they focus on addressing challenges that arise due to big data characteristics of IoT domains — viz., the 4V challenges: volume, velocity, variability, and veracity. This thesis, in contrast, focuses on what we refer to as the data **virtualization** or the fifth "V" challenge. By the term data virtualization, we mean the difference in the abstraction levels between observations generated by sensing devices and their domain-specific semantic interpretation. For instance, a sensor observation may be an image captured by a camera. Its semantic interpretation might be that "John entered the Einstein Building" if, for instance, the camera was situated at the entrance of the building.

In smart space domains, application logic is best expressed at the semantic level, though data arrives at the raw sensor level. We call a system that supports data to be viewed at both levels of abstractions (and provides mechanisms to seamlessly translate concepts, data, and queries across them) as supporting data virtualization. We argue that such a system offers several benefits from the perspective of building smart space applications. First, and foremost, such a system will significantly reduce the complexity of building smart space applications from the perspective of application developers.

To illustrate the benefits provided by data virtualization, we show the challenges faced by

application developers in building IoT applications over large-scale sensor deployments using the following example:

Example 1.1. *Consider a smart campus that captures people's location to support location-based services such as: locating group members in real-time, customizing heating, ventilation, and air conditioning (HVAC) controls based on user preferences, contact tracing by determining who came in contact with whom and when, monitoring occupancy of different parts of a building over time, analyzing building usage over time, understanding social interactions amongst residents and visitors, and keeping track of facilities visited by visitors. While a smart campus requires multiple types of sensors (e.g., HVAC sensors such as temperature, humidity, and pressure sensors), for the sake of our example, we focus on location sensing. In particular, people can be located based on (a) GPS sensors on their mobile device which transmit the GPS coordinates to the system using a client application, (b) camera-based localization in locations where cameras are installed, (c) using connectivity events in the WiFi network using techniques such as [83, 82]. Each of these mechanisms has its benefits and limitations. The GPS-based method works only if the user has actively downloaded an app on their device, and that too, if the user is outdoors. Cameras are typically installed in limited locations and are, furthermore, more time consuming to analyze. WiFi events, on the other hand, provide potentially a ubiquitous localization solution, but might not be as accurate.*

Application developers face the following challenges while developing the localization application mentioned in the example above.

- ***Multiplicity of Sensors:*** Different types of sensors can have overlapping capabilities and could be used to generate the same application-level information. For example, WiFi connectivity events, GPS, and/or camera all can be used to localize a person. The mechanism of how to choose which sensor to use, when, in what context, will have to be part of the application adding significant complexity to the design.
- ***Availability of Sensors:*** The sensing infrastructure of smart spaces may evolve

constantly. New sensors may be added, e.g., part of the campus gets deployed with Bluetooth beacons making a new sensing technology available for locating people. A new algorithm for face detection might be deployed — increasing the set of options. Furthermore different sensors may be available at different locations and at different times. For example, if a user's GPS is off or the user is indoors, the GPS cannot be used to locate the person. Likewise, cameras can only be used in parts of spaces that are instrumented. Or, the sensors may themselves be dynamic, e.g., mobile sensors, and as such, their availability at different times and different locations may vary. Incorporating such complex situations into application logic will make writing code for smart space applications very hard.

- ***Opportunities for optimization:*** Processing data from different types of sensors can have different execution cost. E.g., localization, processing one WiFi connectivity event might take ≈20ms [83] , while analyzing a single camera image might take ≈0.4s [139] (as it will require face recognition code to be run on the image). Both WiFi APs and cameras may cover multiple (possibly overlapping) regions. In such a situation, there are multiple possible ways to localize people and the system should be able to select sensors in such a way so as to minimize the total execution cost and/or to explore a trade off between quality and cost. Furthermore, as the sensors often fail, the system should be able to provide tolerance against sensor failures by automatically moving to available sensors that have capabilities and coverage overlapping with the faulty sensors. Adding the burden of optimizing such costs or tolerating sensor faults to application logic is complex and sub optimal as has been shown in the context of query optimization [77] and database transactions [47] respectively.

A system supporting data virtualization will relieve the application logic from having to deal with sensor capabilities, placement, and availability. Such complexities will now be hidden by the appropriate abstractions supported by the system enabling application writers to

write code almost entirely at the semantic level. Furthermore, data virtualization offers the benefit of extensibility (since new sensors and sensor types can be added without modifying application code), as well as, unravels multiple unique opportunities to optimize data processing by exploiting the semantics of the smart space.

This thesis describes TippersDB, a system designed to support data virtualization for smart space applications. TippersDB offers several unique features discussed below:

Semantic Abstraction. TippersDB supports a novel two-tier data model that separates the sensor data from the higher-level semantic data. TippersDB models the physical world/-domain in the same way we model domains in current databases – as physical entities and relationships. The difference is that we are now modeling the dynamically evolving physical world that is *observed* through sensors. TippersDB uses an Entity-Relational Model suitably extended to support the dynamic nature of an evolving smart space for this purpose. In particular, attributes of entities (and relationships between them) may be static or dynamic that may change over time. For instance, a person's name may be static, but his/her location may change with time. Likewise, relationships between entities may be dynamic. E.g., if the system captures the fact of persons entering into rooms, then based on the movement of a person, a new relationship between a person and a room may dynamically emerge. Such dynamic aspects of data are represented using an extended Entity-Relationship (ER) model, referred to as the Observational ER (OER) model, which represents temporal data evolution. In addition to representing data at the semantic level, TippersDB represents data at the sensor level in the form of data streams. It also provides mechanisms for the specification of functions to translate data at the sensor level into higher-level semantic abstraction. Such a layered data model decouples application logic from sensors and alleviates the burden of dealing with sensor heterogeneity from application programming which greatly reduces the complexity of developing smart-space applications.

Transparent Translation. TippersDB optimizes the translation of sensor data to generate

semantic/application-level information. TippersDB associates *observing functions* with the dynamic properties, which observe the "value" of the attribute/relationships through sensors. Also, TippersDB maintains a representation of sensors and their coverage (i.e., what entities in the physical world they can observe) as a function of time. Different functions may differ in the cost of generating the observation and different sensors might be available to monitor a particular entity at a given time. This cost and availability aspect is utilized by TippersDB to optimize the translation of sensor data to generate values for dynamic attributes.

Query Driven Translation. Sensor data translation can be done at ingestion or during query execution. In IoT-based systems, sensors continuously generate data, causing the data arrival rates to be very high. Processing sensor data at ingestion using Streaming systems (e.g., Spark Streaming [137], Storm [127] – often used for scalable ingestion) leads to significant overhead. Therefore, complete sensor data translation at ingestion is not viable. This observation has also been made in several prior works [63, 32]. The alternate strategy of translating sensor data at query time is more suitable. Not only does it avoid the large ingestion time delay, but it also reduces redundant translation of sensor data if the applications end up querying (a small) portion of the data. This query-time translation is also consistent with the modern data lake view architecture where we store and process only data that is needed. TippersDB uses such an architecture and performs query-driven translation by adding translation as an operator inside the query plan, co-optimizing translation, and query processing.

Progressive Query Processing. Although the query-driven translation avoids the ingestion delay, the latency of queries are now increased since the sensor data gets translated/processed when the query arrives. This causes, the analysts/users to wait longer to get the results of the query. To overcome this limitation of query-driven translation, we develop progressive query processing techniques that progressively translate data and provide early results to the user. To progressively return results TippersDB returns results of lower quality

or at a coarse level early and keeps returning results of better quality and at a finer-grained level as more and more time passes. Consider, for instance, our localization example. LOCATER [83] shows that the location determination from the WiFi connectivity data at the region level (coarse-grained) can be done very fast (20ms), however, finding the room level (fine-grained) location takes a much longer time (200ms). TippersDB uses such a difference in cost of translation at different levels of hierarchy to return early but lower quality results.

Thesis Contributions

This thesis makes the following main contributions in the context of developing a system to ease smart space application development:

- We introduce **SmartBench**, a benchmark that we developed to test different database systems in their suitability for smart space applications. SmartBench, focuses on workloads resulting from (near) real-time applications as well as longer-term analysis of IoT data. SmartBench, derived from a deployed smart building monitoring system, is comprised of; 1) An extensible schema that captures the fundamentals of an IoT smart space; 2) A set of representative queries focusing on analytical tasks; and 3) A data generation tool that generates large amounts of synthetic sensor and semantic data based on seed data collected from a real system. We present an evaluation of seven representative database systems and highlight some interesting findings that can be considered when deciding what database technologies to use under different types of IoT workloads. Using the results of the benchmark we show that the lack of data virtualization in existing database systems, makes them difficult to use with large-scale IoT systems, and therefore, new architectures that support data virtualization are needed.

- We introduce the **TippersDB data model**, which provides data virtualization through semantic abstraction. The data model hides the complexities of sensor infrastructure and sensor data processing codes from the application developers and

provides them an interface to write applications on top of higher-level semantic data.

- We present the **design and architecture of TippersDB**. We describe how the data model is realized using a middleware-based system. We describe how TippersDB translates sensor data to generate semantic data. We show how TippersDB reduces the number of translations by removing redundant translations using its query-driven translation mechanism and several other optimizations.

- We present **progressive query processing** techniques to further reduce query latency and to provide early results to the users. We describe the semantics of progressive query processing and ways to provide progressive answers. We show how TippersDB selects data to be translated, exploits hierarchical data types (e.g., location), and computes incremental answers.

TippersDB has been developed as the backend of the TIPPERS system [92] which has been adopted and deployed by multiple schools and campuses including UC Irvine, Ball State University, Honeywell and Walnut Village. At UC Irvine, it was also used for real time monitoring of occupancy of different regions [40] during COVID-19. TIPPERS was also part of the Trident Warrior(TW) exercises in year 2019 and 2020 where it was deployed on navy ships [35].

The rest of this thesis is organized as follows. In Chapter 2 we explore the previous work related to TippersDB. In Chapter 3 we present the SmartBench benchmark. Chapter 4 presents the TippersDB's layered data model. In Chapter 5 we describe realization of TippersDB's data model and its query-driven translation mechanism. In Chapter 6 we discuss progressive query processing techniques to reduce query latency. In Chapter 7 we show a few case studies based on TippersDB's data model. Finally, Chapter 8 concludes by summarizing the contributions and the possible future extensions to the work presented in this thesis.

Chapter 2

Related Work

In this chapter, first, we review previous work on designing specialized data models, frameworks developed to model sensors, and data models designed to manage temporal data. Next, we highlight the current state of the art systems that are used by IoT applications in storing and processing sensor data. We describe how these systems can not suffice for developing efficient and easy to program IoT applications. Next, we discuss previous work done on query-driven data processing in different domains. We also review, techniques that were developed in the past to provide users with early results(possibly of lesser quality) that incrementally get better. Finally, we review several database benchmarks that were developed in the past in the context of their applicability to IoT applications.

2.1 Specialized Data Models

Database systems are widely used in a variety of application contexts as they provide ways to efficiently store/process data and support a data model (e.g. relational data model) that allows easy manipulation and retrieval of data. Database systems provide a general technology

that different applications can use to manage their data. However, for application domains having large and complex business logic, directly writing applications on top of a database system using the database's data model becomes challenging. Application developers are well versed with the application logic but, in general, do not have the specialized knowledge of tables and the complex network of relationships between them in the underlying database system. In the past, application specific middleware systems have been developed on top of existing database systems for several application domains (e.g., **Enterprise Resource Planning (ERP) systems** [25, 22], **Customer Relationship Management (CRM) systems** [24], and **Financial systems** [23]). These systems provide higher-level models that capture the essence of the application domain. For instance, a CRM model may have an in-built data model for customer contact, buying history, etc. Such a data model is then appropriately mapped, stored, and processed on top of a database system by the middleware-based system. This layered approach has been successful and widely adopted since it leverages the power of database systems to manage, store, and process data while supporting higher-level models to make application development easier. Given the importance and emergence of the Internet of Things, sensor-based systems, and smart spaces, we believe it is time to think of specialized middleware to provide appropriate abstractions to build applications on top of sensor data.

IoT frameworks: Several IoT frameworks [128, 76, 50, 107] have been proposed in the past to ease IoT application development. However, these frameworks only abstract out communication and management of IoT devices. Although these frameworks make the task of fetching data from different types of sensors easy, they do not provide any semantic abstraction over sensors to the application developers, who still have to translate sensor data in the application code itself. Similarly, industrial systems like Nest [10], AWS IoT [4] only allow easy access to the sensors and sensor data but fails to provide any semantic level abstraction. Furthermore, there has been prior work in modeling sensors as devices that generate observations about measurable properties. The most popular of such sensor

representations are provided by SensorML [46] and the W3C Semantic Sensor Network (SSN) ontology [70]. Such representations can be used to store metadata about sensors, however, they still do not fill the gap between the applications and the sensor data.

Temporal data model: As the environment is constantly evolving, it is important for an IoT system to model constantly changing phenomena. One of the specialized data models that were developed to model changing data is the temporal data model. In the past, temporal database systems have been developed [114, 126] that model the dynamically changing world and maintain all changes. Temporal databases, along with the current state of the data, maintain all historical values (with the time instances when the values were updated), allowing users to write queries on any historical database state. Temporal databases, to differentiate real world changes with database updates, maintain two types of timestamps [113]. *Valid time* represents when an event occurred in the real-world and *transaction time* represents the time when the database recorded the change. There are also works in the past [89, 36, 59] that explored different ways to map temporal data models to the relational model. As we will see in Chapter 3, the TippersDB's mapping of its Observable ER model to relations is inspired by the previous work done on the temporal data models and its mapping to the relational model.

Spatio-Temporal database systems: Modeling spatial extent and the movement of different entities in the spatial extent is an important aspect of a smart-space IoT system. In the context of database systems, the concept of *moving objects* has been studied in the past. Several models to maintain moving objects in a relational database systems have been proposed, for example, authors of [58, 57] have proposed an abstract data-type based approach. Although these spatio-temporal data models are very useful to represent the movement of different entities in a spatial region, there is a need for a system that can allow the developers to write their queries on these spatio-temporal data models without worrying about how the actual spatio-temporal data is being computed from the sensor data. For example, location of

people, dynamic coverage of moving cameras can be modeled using spatio-temporal database systems, however, these database systems can not directly compute a person's location at a given time by finding the appropriate camera.

2.2 Data storage and processing systems

In recent times, several database systems and technologies have started advertising themselves as suitable for IoT data management. In this section, we review some state of the art database technologies that are used by IoT based systems.

Timeseries database systems: One of the main aspects of an IoT system is to efficiently store and query a large amount of sensor data arriving at a very high velocity. At present, timeseries database systems such as [33, 95, 20] are the leading systems in storing sensor data. These are specialized database systems optimized to store and query a large amount of sensor data. Timeseries database systems support a very high ingestion rate, compression, columnar storage, time based sampling, and summarization/aggregation. Although timeseries database systems are ideal candidates to store sensor data, IoT applications cannot be directly supported by such systems as these systems are not good to store traditional relational data, and most of them do not even support full SQL. Furthermore, they do not provide any mechanisms to translate sensor data.

Streaming systems: In order to process/translate sensor data as it arrives, IoT systems often use stream processing systems. Streaming systems such as Apache Kafka [3], Storm [127] and Flink [48] were developed in the past decade that provide a scalable way of processing data prior to its ingestion. They provide a distributed, high-throughput, low-latency platform for handling high velocity data streams and therefore, are used to develop scalable data ingestion pipelines in various domains. Many of these systems (e.g., Spark [137], Apache

AsterixDB [29]), allow users to write UDFs (user defined functions) to process streaming data prior to ingestion. Previous works such as [132] have developed methods to improve stream data processing by exploiting parallelism and batching data/operations. While such techniques and systems have improved data processing during data ingestion, they still cannot scale when the data processing to be done uses much more complex and computationally expensive functions (as can be the case in IoT applications) and therefore, techniques that process data at the query time are needed.

Data warehouses: Another possible way to translate sensor data is through an offline process where the sensor data for a day/week/month is collected and processed using the traditional extract-transform-load (ETL) [129] approach. In this approach data is periodically collected, processed, and then loaded into a data warehouse [1, 13] to be used by the applications. Typically, such systems, pre-compute summaries/aggregates at different granularities to make the queries faster. The ETL approach can be used for those IoT applications that perform long term analysis tasks and do not require real-time data. However, the ETL approach suffers from several limitations: (1) a long delay between sensor data arrival and data available for analysis, (2) Redundant translations if applications only use a small portion of the data and (3) new sensor data processing function cannot be added after the sensor data is already translated and ingested into the data warehouse.

HTAP systems: To develop a holistic IoT system that can support both real-time as well as analytical applications, we need systems that can achieve a high sensor data ingestion rate as well as can answer complex analytical queries quickly. For this purpose, in the past few years, several Hybrid Transactional/Analytical Proccessing (HTAP) systems [39, 102, 87] have been developed to support real-time analytics on the data as it arrives. The design choices made by the HTAP systems, allowed them to support fast transactional workloads with insertions/updates and yet support near real-time analytics on the incoming data. HTAP systems support a flexible data storage model, with the incoming data getting stored in

row format and subsequently getting moved/stored to a columnar format (often maintained in the main memory). The initial row based format allows HTAP systems to provide good performance on inserts/updates whereas the columnar format (which is highly compressible) allows them to support fast analytical queries (e.g., computing aggregates). This eliminates the need for pre-computing aggregates which is done in the case of data warehouse based systems. While HTAP systems today have focused on the storage layer and optimization of queries, based on the new storage models, they have not considered supporting expensive transformations that might be necessary to prepare the data for analysis, as is the case for IoT applications domain.

Data lake architectures: Our work on query driven translation has a motivation similar to that of data lakes [74]. In the data lake architecture, the incoming data from various data sources are initially just stored into a large data lake system and the analysis and processing of the data is performed later at the read/query time. Such systems are often realized through an extract-load-transform (ELT) pipeline, instead of the ETL pipeline.

2.3 Query Execution

As we have shown in Chapter 1, it is not feasible to process sensor data arriving at a very fast pace using computationally complex functions. Similar issues have been encountered in other domains as well, which has motivated many researchers to develop query-driven techniques of data processing. In this section, we review the past work done to integrate query execution and data processing. We will also review work done in the past to reduce the increase in latency caused by query-driven data processing.

Query driven data processing: Several areas of database research have developed query driven techniques to reduce redundant work. In the data cleaning context [63, 32] have

developed query driven approaches and used query context to eliminate objects that cannot satisfy the query predicates. More recently, EnrichDB [61] and JENNER [62] have introduced query driven data enrichment. However, EnrichDB is designed for a different purpose i.e., to integrate machine learning based inference functions into the database system. Given its design criteria, its focus has been on functions such as classification applied to a row at a time. Sensor data translation, in contrast is very different. It may require accessing data from several tables at the same time, choosing a plan involving sensors of different types to generate semantic data. Therefore EnrichDB cannot be directly used for sensor data translation.

Approximate query processing Query-driven translation can potentially increase the query latency and the wait time for the end user since the data gets processed during the query itself. To avoid making the user wait for the results TippersDB provides users with early but possibly lower quality results. This approach of reducing query latency by providing lower quality results early is similar in motivation to the approximate query processing (AQP) systems. Such systems were developed for applications that need complex analytical queries to be answered quickly on large volume of data but can tolerate bounded inaccuracies in the answers. AQP systems provide an approximate answer with an error bound to give a quick insight into the data. AQP systems can be categorized into two types. Offline AQP systems [27, 100, 54] that maintain a set of pre-computed samples of different sizes consisting of different column sets and rows. Given a query with an error or latency bound, the system finds out the best samples for the query. In contrast, online AQP systems [109, 138, 135] samples data at the query execution time itself. Although AQP systems reduce user wait time when running queries on large data, they cannot be directly used to reduce query latency caused due to the translation of sensor data at query time.

Progressive data processing: TippersDB approach of providing incrementally improving results to the users is motivated by previous work done in the context of *progressive query*

processing. Progressive query processing techniques not only provide quick results to the user, they also optimize the overall execution of the task by learning from the execution of the task itself. EnrichDB [61] and JENNER [62] developed progressive query processing techniques in the context of query driven data enrichment. They divide the query execution into epochs and in each epoch they select a set of objects(along with enrichment functions) to be enriched such that the overall increase in the quality of the query answers is maximized. Previous work such as [86, 99, 30] have used progressive techniques in the context of entity resolution. Similarly, progressive approaches have been used in online schema matching [88, 108], and probabilistic databases [52, 110, 111]. Systems built for interactive visualizations e.g., Cloudberry [78] have also used progressive approaches that provide users with quick results at a coarser level. However, similar to AQP systems, Cloudberry tries to solve the problem of doing visualization on large datasets and does not consider the issues arising from processing datasets using computationally expensive functions.

Incremental View Materialization (IVM): One of the important requirement from a progressive query processing technique is to be able to generate delta answers (answers that got added/deleted) without minimal overhead. Incremental views [43] are a very good candidate for generating delta results and are widely used in generating query answers as the underlying data gets updated. IVM computes and applies only the incremental changes to the materialized views. Suppose that view $V(Q)$ is defined by query Q over a state of base relations D. When D changes $D' = D + \delta D$, we can get the new view state $V'(Q)$ by from D, δD and Q using $V'(Q) = V(Q) + \delta Q(D, \delta D)$. Several works in the past have proposed mechanisms for efficiently updating materialized views e.g., DBToaster [80], LINVIEW [96]. More recently differential data flow system [90] have been developed that provide efficient state management techniques to speed up delta computation. Furthermore, intermittent query processing systems [120, 122, 121] and Tempura [133] have developed techniques to support efficient ways of maintaining and computing delta answers while minimizing resource consumption. Efficient delta computation has also been studied in data flow systems [93,

28, 90] that provide efficient state management techniques to speed up delta processing. All these works are complimentary to TippersDB as we use incremental views as a building block in our progressive query processing approach.

2.4 Benchmarking

Several benchmarks have been developed to test database systems from diverse perspectives. The most widely used benchmarks are TPC-H [51] and TPC-DS [94], which were designed to test a database system's analytical data processing and data warehousing (OLAP) performance under different data workloads. TPC-C [16], 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 [104] 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 [37] 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 [112] 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. [53] 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) [34] and StreamBench (distributed streaming) [84], are focused on testing the performance of real-time data insertion/processing and queries. In contrast, big data benchmarks like BigBench [60], BigFUN [103], HiBench [72], and BigDataBench [131] compare systems on their ability to process a large volume of heterogeneous data. BigBench [60] 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 [103] is a benchmark based on a synthetic social network scenario with a semi-structured data model. HiBench [72] and BigDataBench [131] 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 [37]

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., [66, 116, 71, 123, 67]). Gray et al. [66] developed ways to generate large amounts of data in parallel with different user-provided distributions. MUDD [116] and PSDG [71] scan a complete user-provided dataset to compute distributions and dependencies in the data and then generate synthetic data with the computed distributions. UpsizeR [123] uses clustering algorithms to find foreign key based correlations but assumes that non-key attributes depend only upon the key attributes. Chronos [67] 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.

Chapter 3

SmartBench Benchmark

Central to IoT applications is the database management system (DBMS) technology that can represent, store, and query sensor data. Given the importance of IoT, a multitude of DBMSs, whether they be standard relational systems, key-value stores, document DBs, or specialized systems such as time series stores, have begun branding themselves as being suitable for IoT applications.

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

Finally, the sensor data is often too low-level for being useful for the final applications directly

and needs to be enriched/translated into higher-level semantic inferences. For instance, to provide personalized thermal comfort, an application may need to know the occupancy of different parts of a building, which may need to be inferred from diverse sensors including camera data, WiFi connectivity information, door sensors, and bluetooth beacons. Such interpretation is often performed using complex machine learning algorithms [85] outside the database system.

While different DBMSs provide different functionalities, none provide a comprehensive solution to the above challenges. Big Data management technologies built on top of cluster computing frameworks (*e.g.*, Hive [124], SparkSQL [38]), provide efficient ways to run complex OLAP queries on large amounts of data spanning multiple machines. However, they fail to do well on fast data ingestion, simple selections, and real-time queries. In contrast, stream processing systems like Apache Kafka [3], Storm [127], and Flink [48] 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 [11], make use of indices and better join and aggregation operator implementations, but they do not scale well when the data becomes large and inherently do not support storage of heterogeneous data. Document stores like MongoDB [9], Couchbase [5], and AsterixDB [29] 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 [19] and GridDB [7], provide fast ingestion rates and fast selection on time ranges but they fail to support complex queries. IBM DB2 event store [8] is an in memory database that provides a very fast ingestion rate along with streaming data analysis capabilities.

The DB community has traditionally relied on benchmarks to understand the trade-offs between different DBMSs (e.g., the TPC-H [51] benchmark has been widely adopted by both industry and academia). In recent years, IoT DB benchmarks have begun to appear (e.g., [34, 37, 53, 84, 104, 112]) but the focus of such benchmarks has been on comparing

systems based on fast ingestion of streaming sensor data. While ingestion is critical, IoT settings also require DBMSs to support one-shot queries over IoT data both for real-time applications and for data analysis.

We present *SmartBench*,[1] a benchmark focusing on evaluating DBMSs for their suitability in supporting both real-time and analysis queries in IoT settings. SmartBench, derived from a real-world smart building monitoring system (currently deployed in several smart buildings of the campus of the University of California, Irvine [91]) is comprised of an IoT data model, a set of representative queries, and a tool to generate synthetic IoT datasets. The schema captures the main concepts related to IoT environments and is, thus, extensible to different environments. It supports heterogeneity of sensors by using a semi-structured representation that can be mapped to the data model supported by the underlying DBMS. It also presents several mappings of such a representation to underlying DBMSs.

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

While SmartBench is based on a smart building context, it can be applied to other IoT systems as well because of its flexible schema which divides an IoT space into data, domain and device layers. We provide detailed performance results and an analysis of the performance of seven representative database systems (covering different technologies such as time series

[1] See `http://github.com/ucisharadlab/benchmark` for SmartBench (code/data generation tool).

and specialized databases, relational DBs, document stores, and cluster computing frameworks). We test scalability (both scale up and scale out) of different DBMSs by comparing their performance under different data loads for a single node as well as a multi-node setup. From the results we have affirmed the intuition of a lack of a silver bullet. However, we have seen that some issues of specialized databases with respect to more traditional approaches (like the lack of support for complex operations) can be mitigated through external code that for typical IoT operations can perform adequately. We conclude with some interesting key observations that can help IoT system developers select DB technology(ies) for their data management needs and guide future DB developers to support such needs.

3.1 SmartBench Benchmark

We begin discussion of SmartBench by first highlighting the goals that guided our design. SmartBench is designed to explore data management needs of pervasive sensing environments, where a single smart space may house a large variety of sensors ranging from video cameras, microphones, thermostats, beacons, and even WiFi Access Points (which can sense which devices are connected to them). Sensors may overlap in the type of physical phenomena they can sense E.g., one can determine the occupancy of a location using connectivity of devices to a WiFi access point. Occupancy can also be estimated based on temperature or power draw (e.g., number of power outlets connected to devices). It can also be estimated using the number of times a motion sensor in front of an entrance trips, or by using people counters. Each of these sensor modalities have their advantages and disadvantages from the perspective of applications.

Challenge 1: DB technology must provide mechanisms for the dynamic addition/deletion of new sensor types without requiring the system to stop. The system must support the ability to store and query data from sensors of newly added types. In designing our benchmark,

we took into account such heterogeneity by including a general and extensible schema that enables generation of different types of sensors. The schema is mapped to the underlying DBMS based on the data model supported by the system.

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

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

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

Challenge 3: DBMSs need to support a wide range of applications with different demands ranging from simple queries over recent data to fairly complex analysis of historical data. Also, such queries might involve querying raw sensor data as well as abstractions built on

top of raw data. This can result in complex queries on the higher-level semi-structured data model, where typical DB operations like joins and aggregation can be desirable. In our benchmark we have included queries of this kind, including those required for both real-time and analytical applications.

3.1.1 SmartBench Schema

The schema used in SmartBench is based on the Observable Entity Relationship (OER) Model. OER is an extensible model that allows incorporating new/heterogeneous types of sensors, observations, spaces, and users. SmartBench's complete data model is specified in Appendix A. In the following, we highlight key entities and concepts in a smart space categorized into three interrelated layers.

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

Each sensor has attributes *type*, which dictates the type of observation the sensor generate, and *coverage*, which corresponds to the spatial region in which the sensor can capture observations. For physical sensors, coverage is modeled deterministically [136] and simplified as a function of its location - e.g., the coverage of a camera is its view frustum. For virtual sensors, coverage is a function of the coverage of the sensors used as their input - the exact function depends upon the specificity of virtual sensor. For example, the coverage of the

presence detector (that determines location of people in the immersed space) based on camera inputs is the union of the view frustums of all the input cameras. Each physical sensor has a *location* and each virtual sensor has a property, *transformer code*, corresponding to the function applied to input sensor data to generate semantic observations.

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

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

Domain Layer. This layer is comprised of the spatial extent of the smart space and information about subjects who inhabit it. Both of these concepts are inherently hierarchical, with the hierarchy representing granularity (e.g., a campus may consist of buildings, which have floors, which, in turn, are divided into rooms and corridors; likewise people are divided into groups – such as faculty, students, etc.). Domain entities have associated attributes which are classified as *static* or *dynamic*. These attributes, introduced in the W3C SSN ontology [18], model relevant and high-level information that smart applications would require about the space itself (e.g., its structure and functioning) or people within. Static attributes (e.g., the name of a room or a person, the type associated with rooms such as

meeting room/office) are typically immutable, while dynamic attributes (e.g., the occupancy or temperature of a room, the location of a person) change with time. Dynamic attributes are *observable* if they can take (physical or virtual) sensor input to determine their value. Observable attributes are mapped to sensors using a function (*inverse coverage*) that given an entity (i.e., users/spaces), the type of observations required, and time, returns a list of sensors that can generate observations of the required type observing the required entity at the required time. For instance, *Inv_Coverage(Room 2065, t1, occupancy)* will return a set of (virtual) sensors that can output the occupancy of Room 2065 at time t1.

To implement the above model in a DBMS, one needs to map the appropriate concepts (viz., domain entities, sensors, observations, etc.), into database objects. These mappings are database dependent, as we will explain in Section 3.1.4.

3.1.2 Queries & Insert Operations

The benchmark consists of twelve representative queries motivated by the the need to support diverse applications as mentioned in Section 3.1. 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 next four queries are on higher-level semantic data (viz., on the presence of people in the space and occupancy of such spaces) and are chosen to represent different important functionalities provided by applications. The last two queries capture window-based operations and continuous queries. Almost all of the queries have a time range predicate specified, as would be expected of queries in the IoT domain. Below, we describe the queries and rationale behind their selection.

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

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

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

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

• **(Q6) Statistics($S \subseteq Sensors$, A, F, t_b, t_e)**: retrieves statistics (e.g., average) based on functions specified in F during the time range $[t_b, t_e]$. The statistics are generated by first grouping the data by sensor, and further by the value of the attributes in the list A. For

instance, a query might group observations on sensor_id and day (which is a function applied to timestamp) and compute statistics such as average. Such a query is important to build applications that provide the building administrator information about the status of sensors (e.g., if the sensors are generating too much data or none at all, discovery of faulty sensors).

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

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

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

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

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

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

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

• **(IE) Insert&Enrich**($s \in Sensors$, t, $payload$, $params$): takes as input (raw) sensor observations and mimics execution of an enrichment pipeline to generate and insert semantic observations. Such a pipeline typically consist of a sequence of operations. Based on the type of data, it first identifies the set of enrichment functions that need to be invoked. Each of those functions, may result in one or more queries over both metadata or data (e.g., queries of types Q1-Q5). For instance, to transform a WiFi connectivity event (indicating that a given device connected to a specific AP at a given time) a query of type Q1 may be executed to determine the location of the AP following which sensor data from neighboring APs about the device (or other devices) might be accessed (queries of type Q3-Q5) to predict the semantic location (e.g., the room) in which the owner of the device might be. Likewise, for a camera sensor, image segmentation and face recognition may need to be performed to determine the identity of a person in the view. Since enrichment functions (e.g., face recognition, location determination, etc.) are performed outside of the DBMSs [132], the choice of specific enrichment function does not influence suitability of a DB technology to IoT application. We simply model the I&E pipeline as a sequence of queries followed by a busy wait (to mimic an execution of the enrichment function), followed by an insertion of the semantic observation. The specific queries generated in the pipeline and the wait time are *parameters* which are input to the IE function.

The set of queries listed above represents an important functional aspect of building smart space applications that have been motivated by the campus-level smart environment that

we have built at UCI. Note that these queries represent sample IoT data management tasks involving operations including selections, joins, aggregations, and sorting.

3.1.3 Data And Query Generator

Benchmarks typically offer tools to generate datasets of varying size to test systems in different situations [66, 116, 71, 123, 67, 37]. SmartBench's data generation tool[2] uses seed data from a real IoT deployment (our University building) including sensor data and metadata about the building and sensors, which is able to scale up/down to create a synthetic dataset. The tool can scale the IoT space (i.e., number of rooms, people, sensors) as well as sensor data (i.e., number of observations per second, time period during which the sensors produce data, etc.). The tool preserves temporal and spatial correlations in the seed data to support realistic selection and join queries. We developed our data generation tool, rather than using a uniform distribution for every attribute independently, since we aimed at comparing different DBMSs with more realistic data (for a smart space setting). The data patterns (e.g., variation of the occupancy values between the day time and the night time, variation in the number of WiFi connections at different points of time) affect the execution time of queries based on the query parameters. To maintain a fair comparison, we execute exactly the same instances of queries on the same data for each DBMS. We do not aim at characterizing types of smart space data based on the patterns it contains.

In addition, the tool also generates an actual query workload based on the query templates described above. Its input is the already generated metadata and a configuration file containing different parameters. All of the queries except Q1 and Q2 include a time range based filter $[t_b, t_e]$. t_b is selected at random from the range $[T_b, T_e]$ where T_b and T_e are the minimum and maximum timestamp of the entire observation dataset, respectively, and t_e is selected at random from the range $[t_b + \delta_a, t_b + \delta_b]$ where values for δ_a and δ_b are provided in the

[2]An extended explanation of the tool, including configuration parameters supported, is available in [12].

Table 3.1: Different DBMS and their capabilities.

DBMS	**Sec. Indexes**	**Joins**	**Agg.**	**Column/ Row Store**	**Structured/ Semi-Structured**	**Compress.**	**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.

configuration file. List based query parameters, e.g., the list of sensors in query Q4 and Q6 and the list of locations in query Q10, are generated by randomly picking (without replacement) n elements from the already generated metadata; n itself is selected at random from the range $[n_a, n_b]$, where n_a and n_b are provided in the configuration file. Scalar parameters like user u in query Q8 are selected at random from the available values in the metadata.

3.1.4 Model Mapping

There are several ways in which the schema in Section 3.1.1 can be represented in the underlying DBMS. We explore multiple mappings of IoT data to the underlying databases, which are broadly characterized based on how sensor data is stored in the database: *(A)* Single table for observations from all sensors; *(T)* Multiple tables with one per sensor type;

and *(S)* Multiple tables with one per sensor instance.

A given mapping approach can be simple and reasonable to apply on some database systems, while difficult or unnatural for others. With PostgreSQL, mapping T is straightforward, while mapping A can be applied using its JSON data type (although not intuitively since the JSON type is stored as a BLOB). Mapping S is not practical since it would create a very large number of tables. Since CrateDB support structured data, mapping T is most natural while A can again be applied using a JSON data type since CrateDB internally stores 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 3.1 for examples of both) – a *Sub-Document Model (A1)*, where each observation is stored as a nested document by embedding related data together (however, since complete nesting can create very large documents, we only nested those attributes that can help in reducing the number of joins), and a second *Normalized Model (A2)* where each observation is stored as a fully normalized document, where every relationship is modeled as a reference between two documents.

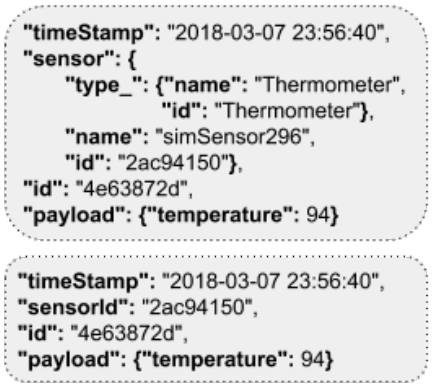


Figure 3.1: Mappings for document stores (A1 and A2).

Timeseries databases such as InfluxDB and GridDB do not support JSON and hence cannot support mapping A directly. However, as mentioned in Section 3.2, 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 PostgreSQL database whenever it is required by the application.

3.2 Database Systems

To evaluate a wide range of database technologies in supporting analytic workloads on IoT data, we broadly classify systems into four categories: traditional relational database systems, timeseries databases, document stores, and cluster computing based systems. For our experiments, we selected representatives from each category to cover a wide range of data models, query languages, storage engines, indexing strategies, and computation engines (see Table 3.1). Our main considerations in selecting a particular DBMS were that it: (a) Provides a community edition widely used for managing and analyzing IoT data at many institutions; (b) Is popular based on its appearance on the DBEngine website [6]; or (c) Is advertised as a 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

Table 3.2: Summary table with pros and cons of different database technologies.

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

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 a 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 [6]). Along with the typical requirements for handling time series data, InfluxDB provides support for various built-in functions that can be called on time series data. It has support for retention policies to decide how data is down sampled or deleted. It also supports continuous queries that run periodically, computing target measurements. (b) *GridDB*, which is a specialized time series database. We selected GridDB because it has a unique key-container data model. The data in the container has a fixed schema on which B-tree based secondary indexes can be created. The container is synonymous with a table in relational database system on which limited SQL functionality is available. GridDB, however, does not provide support for queries that involve more that one container, disallowing aggregation over more than one time series. Containers can be either general purpose or time series only containers. Time series containers have a time stamp as the primary key and provide several functions to support time-based aggregation and analysis.

Document stores. We selected two document stores as representatives of this category for our evaluation: (a) *MongoDB*, which is one of the most popular open source document stores that supports flexible JSON-like documents that can directly be mapped to objects in applications. MongoDB provides support for queries, aggregation, and indexing. It has sharding

and replication built in to provide high availability and horizontal scaling. MongoDB supports multiple storage engines. In this study, we used MongoDB with WiredTiger, which is the default storage engine and supports B-tree indexes and compression (including prefix compression for indexes). (b) *AsterixDB*, which, much like MongoDB, stores JSON-like documents. It, however, supports many more features including a powerful semi-structured query language SQL++[97] (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*, which is advertised as a SQL database for timeseries and IoT applications but does not fit the criteria for a specialized time series database system. CrateDB supports a relational data model for application developers but internally stores data in the form of documents supporting JSON as one of the data types. However, along with storing documents as is, CrateDB stores data in columnar format as well. Its distributed query engine provides full SQL support along with other time based

functions. CrateDB is built on top of ElasticSearch [65] and therefore naturally supports inverted indexes, although it has no support for B-tree based indices.

Table 3.2 lists our apriori expectations regarding the impact of the DB technology choice on the performance of different queries based on factors such as row versus column based storage, support for LSM/TSM trees, temporal predicates, indexing mechanisms, query optimization and processing. The last column in the table lists our expectation on how each technology impacts different aspects of the benchmark. For instance, the choice of underlying data model (semi-structured/structured) would affect how SmartBench data model is mapped to the underlying data model supported by the database system, and the usability of the system.

3.3 Experiments And Results

Dataset and queries. We used the data generator tool with seed data from a real IoT data management system, TIPPERS [91], deployed in the DBH building at UC Irvine. DBH is equipped with various kinds of sensors including 116 HVAC data points (e.g., vents and chillers), 216 thermometers, 40 surveillance cameras, 64 WiFi APs, 200 beacons; there are also 50 outlet meters that measure the energy usage of members of the ISG research group. The TIPPERS instance has been running for two years and has collected over 200 million observations from these sensors. This data also includes higher-level semantic information generated through virtual sensors about the presence of people within the building and occupancy levels.

We specifically used as seed TIPPERS data for one week from 340 rooms, observations from three types of sensors – 64 WiFi Access Points, 20 plug meters, and 80 thermometers– and semantic observations about location of people and occupancy of rooms. With this

Table 3.3: Dataset sizes.

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

seed we generated three datasets with different sizes (see Table 3.3). The query parameters were also generated using SmartBench's generation tool. We controlled the query selectivity by restricting the time range to be 1 $day < t_e - t_b < 4$ $days$ for the base experiments (larger periods of time were used in an experiment to test the impact of query selectivity on the systems' performance). For each benchmark query template, we generated 25 query instances with different parameters. We ran each generated query instance on every DBMS sequentially and their average execution time along with the variance is reported.

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

3.3.1 Single Node Experiments

We tested the DBMSs using the datasets in Table 3.3 on a single node (Intel i5 CPU 8GB RAM, 64 bit Ubuntu 16.4, and 500 GB HDD).

Experiment 1 (Insert Performance - Table 3.4): We compare the insert performance on a hot system (with 90% data preloaded). We used single inserts for GridDB and InfluxDB, batched prepared statements for PostgreSQL, CrateDB, and SparkSQL, batched document inserts for MongoDB, and socket based data feeds for AsterixDB to insert the 10% insert test data. The batch size used is 5,000 rows/documents which amounts to 5 minutes worth of observation data for the large dataset. WAL is enabled for all the systems and the default configuration is used for compression in the systems that support it. We did not perform insert performance tests on SparkSQL since it does not manage storage by itself (instead it depends on external sources – Parquet files on HDFS in our case). Table 3.4 shows the size of the dataset after ingestion and ingestion throughput.

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

AsterixDB performs better than MongoDB due to its support for LSM storage even though MongoDB has a slightly smaller database size from insertions due to its row-level compression. PostgreSQL performs better than AsterixDB, even though it lacks LSM storage, due

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

Table 3.4: Dataset sizes after ingestion (including indices) and ingestion throughput for single node and multi node.

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

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

We conducted an additional experiment on insert with enrichment. The IE pipeline (see Section 3.1.2 consists of query Q1 for the sensor specified in IE operator (metadata query),

followed by one of Q3 or Q4 (queries on the sensor data already ingested) chosen randomly. The parameters for Q3/Q4 are generated using the method mentioned in Section 3.1.3, with min and max time interval of 4 and 8 hours, respectively. We chose τ (busy wait to simulate execution of enrichment function) to be 100ms, based on the time taken for enrichment functions in [132] in the context of tweet processing. Sensor data enrichment, e.g., ML functions on images, could even take longer. Even for this relatively modest value of τ, enrichment cost quickly dominates and becomes a bottleneck and the rate at which semantic observations are generated (10,000/second) could easily be sustained by all the systems. Thus, the experiment did not further provide insight from the perspective of comparing across different DB technologies. Nonetheless, the result points to two interesting asides: (a) optimizing enrichment at the time of ingestion in DBs [132] is an important challenge to support real-time applications that need enrichment, and (b) scaling systems requires additional hardware where enrichment can be performed prior to data being stored in the DBMS.

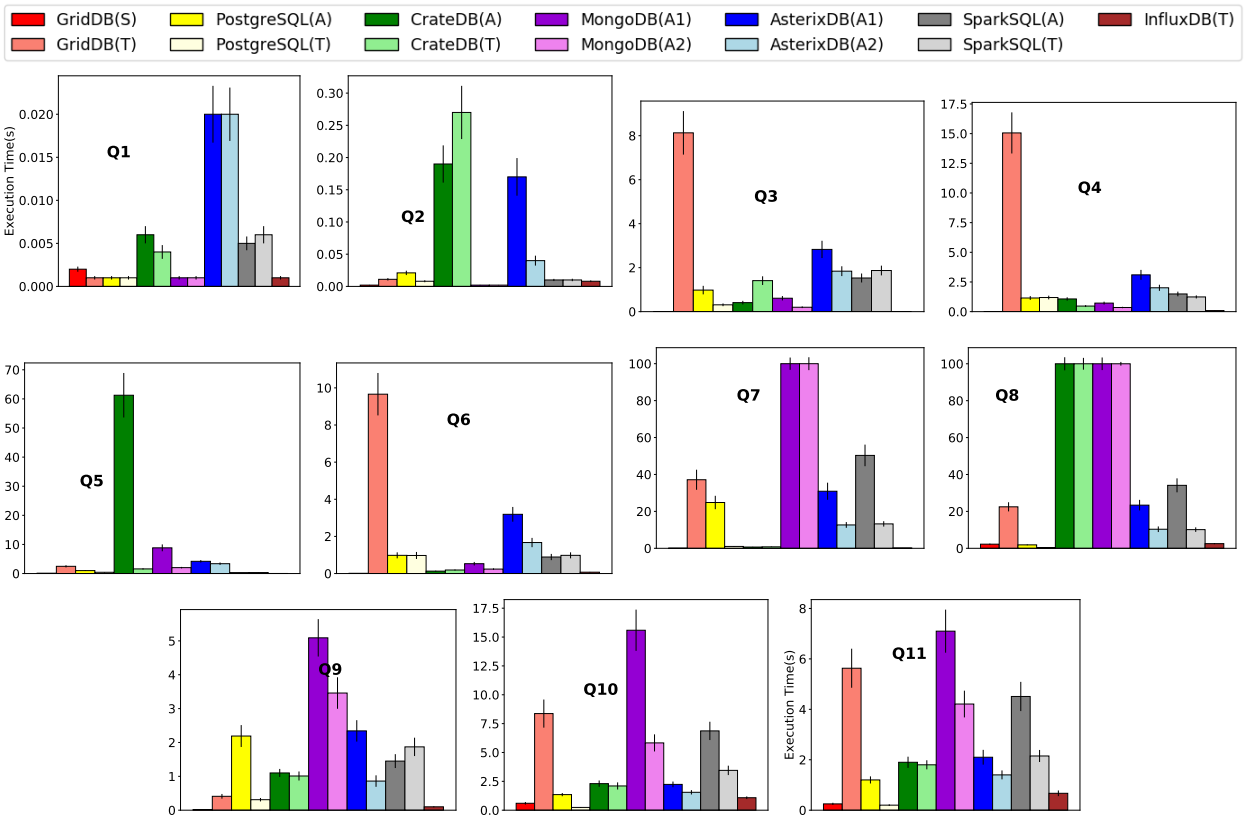
Experiment 2 (Query Performance - Figure 3.2 and 3.3): We compare the performance for the benchmark queries in Section 3.1.2. For all systems, except GridDB and InfluxDB, every benchmark query maps to a single query in the query language supported by the system. For GridDB and InfluxDB, the benchmark queries that involve joins and aggregation cannot be directly executed due to the lack of support for such operations. Hence, we implemented these operators at the application level by pushing selections down, using the best join order, and performing the equivalent to an index nested loop based join. Similarly, for database systems which do not support window queries (i.e., GridDB, InfluxDB, and MongoDB), we implemented an application-level window operator that first fetches data according to the where clause, partitions it based on the grouping key using in-memory hash table, and then sorts the list corresponding to each key based on the ordering attribute. Also, since InfluxDB does not support any way to store non-timeseries data, we stored the building metadata information (users, building info, sensor attributes) in PostgreSQL. Fig-

ure 3.3 shows the average execution time per query on the large dataset, along with standard deviation. Since the same query is run with 25 different sets of parameters (the parameters are the same across DBMSs), we see a significant variance in most queries. The longer the query execution time, the higher the variance, but variance to execution time ratio is larger for queries running within a second. Even with the observed variance, the relative performance among most of the systems can be compared.

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

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

Complex Queries (Q7-Q10). These queries involve complex joins and aggregations that are not natively supported in GridDB or InfluxDB. Hence, their processing requires the application-level implementation of these operators. At low selectivity (date range of only 1

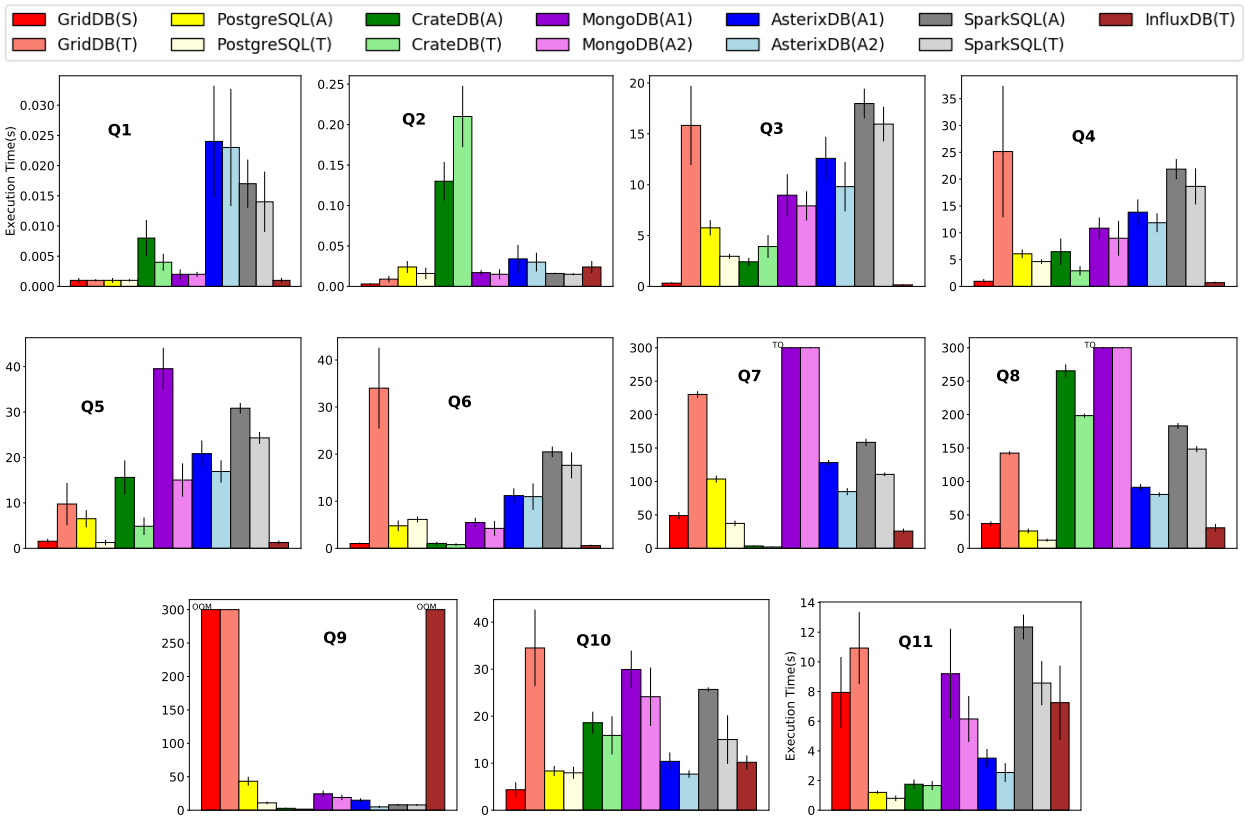


*GridDB + application-level joins. **AsterixDB has a lower bound of 10-20ms for any query due to its operation mode (not optimized for simple or non-cluster queries). ^{OOM}Out of memory error in the application-level code. ^{TO}Timed out.

Figure 3.2: Query runtime (seconds) on small dataset (single node, 5 minutes timeout).

to 4 days), the number of rows to be joined and grouped after all the execution of selection predicates is small and so application-level joins did not cause significant overhead. In fact, except for Q7 and Q8, GridDB and InfluxDB outperformed all the other databases (with native JOIN support) except for PostgreSQL and CrateDB. For Q9 the application level sort-based GROUP BY operator could not store the number of rows fetched in memory what resulted in an out of memory error. GridDB's performance drops significantly with mapping T since it is not able to use the primary index on any of the queries, while InfluxDB performs slightly better than GridDB due to its faster read performance (see Q3-Q5).

CrateDB, a column oriented DBMS (with the benefit of having indexes on all the columns involved in the selection predicate and not just timestamp column), performed best on Q7 and Q9. It took a considerable amount of time on Q8, as it failed to come up with an



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

Figure 3.3: Query runtime (seconds) on large dataset (single node, 15 minutes timeout).

optimized query plan (it tried to do selection after join) and did not perform well on Q10 as the query involved most of the table columns.

All the queries on MongoDB perform better with mapping A2 which suggests that a join (lookup) with smaller metadata tables is many times better than having bigger nested documents in our use case (scanning data based on a time range). Queries that involve joins of two big collections (e.g., queries Q7 and Q8) timed out since the join (lookup) operator in MongoDB is limited in functionality and it failed to push selections down in its aggregation pipeline. AsterixDB supports full SQL functionality, has a more advanced query optimizer, and better JOIN support compared to MongoDB which made its performance much better. PostgreSQL outperformed AsterixDB as the latter has to scan rows which are comparatively larger in size and the former came up with a better query plan since it has a more mature

optimizer and it stores statistics about the data. SparkSQL, even with columnar storage (parquet files on HDFS), did not perform well since it has to scan the entire dataset for all the queries due to the lack of support for secondary indexes.

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

Experiment 3 (Application vs. Database Joins - Figure 3.4): Experiment 2 showed that GridDB and InfluxDB, outperformed systems with native join support using application level joins. We compared them further by varying query selectivity levels. We selected PostgreSQL and Q8 to compare native vs. application-level join as comparing across different systems would make it difficult to determine whether the performance is due to the type of join or other factors. To implement application-level joins, we first send a selection query to the outer presence table, and then, for each row in the result set, a selection to the inner presence table.

As expected, native joins outperformed application-level joins (see Figure 3.4) due to the higher overhead of the latter (e.g., multiple independent queries compiled separately). For the native join case, PostgreSQL selected, for low selectivities, a plan consisting of an index scan on the timestamp predicate of the outer presence table followed by a nested loop index join with the inner presence table and, for higher selectivities, a sequential scan of the outer presence table followed by a nested loop index join. For the application-level join case, it selected the index scan for the outer table at low selectivities, changing to a sequential scan

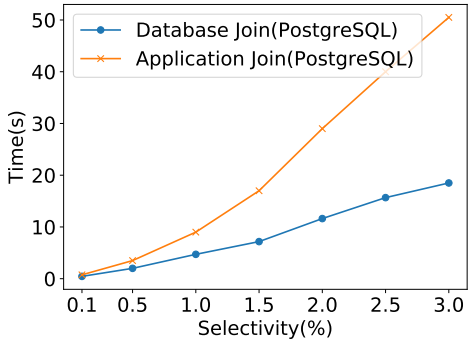


Figure 3.4: Performance of application vs DB joins.

at higher selectivities. All queries on the inner presence table used an index scan on the join column value for all selectivities.

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

Experiment 4 (Effect of Time Ranges - Figure 3.7): Time is a fundamental component of IoT data and queries. We explore further the effect of varying time ranges in queries w.r.t. systems' performance. We chose Q6 and Q8 and varied their associated time ranges to: one day, one week, two weeks, one month, and two months. GridDB, InfluxDB, and CrateDB continue to outperform other systems on Q6 (see Figure 3.7 on the left), although their runtimes increase with the selectivity. PostgreSQL performed as well as the timeseries database systems for low selectivities on Q6, but its performance suffers due to increased secondary index lookups as selectivity increases. SparkSQL performance was not affected since it always performs a table scan. The relative performance of the document stores on Q6

gets worse with increased selectivity due to their comparatively larger record sizes. On Q8 (see Figure 3.7 on the right), AsterixDB, SparkSQL, and CrateDB chose a hash join based approach and took almost the same time for all selectivity values. PostgreSQL, on the other hand, with its mature query optimizer and statistics, chose an index nested loop join, which performed well on low selectivities but dropped with with increase in selectivity. Timeseries databases, using our index nested loop based application-level joins, performed quite well for low selectivity values, but their performance grew super-linearly with increasing selectivity.

Experiment 5 (HDD vs. SSD - Table 3.5): The bulk of our experiments was performed in a cluster with hard disks. We performed an additional experiment to explore the impact of SSDs. To this end, we setup two AWS EC2 instances with same specifications: one with a general purpose SSD and another one with throughput optimized HDD. SSD's provide performance improvement over HDDs on sequential reads and writes, however, they are mainly optimized for random reads and writes and provide order of magnitudes faster IO per second compared to HDDs [79]. We performed insert and query performance test for the single node large dataset (for the best performing mapping in our previous experiments). For all the DBMSs (see Table 3.5) the insert throughput increased because of the superior write performance of SSDs. However, as there were very few random updates and most of the IO performed was sequential and not random, the increase in throughput was limited. For Q6 every system experienced a decrease of execution time due to SSD's higher IOPS. However, the improvement for InfluxDB and GridDB was limited since the data is arranged based on timestamp which implies fewer random updates. For query Q10, all the systems showed a decrease of about 2 times in execution time with SSD since Q10 involves scanning the dataset and therefore the reads are mostly sequential.

CPU and IO Usage: We analyzed the percentage of execution time than went into CPU and IO tasks per query (plots are shown in Figure 3.5). In cases where the query timed out or threw an out of memory error, we analyzed the CPU/IO breakdown before that. For

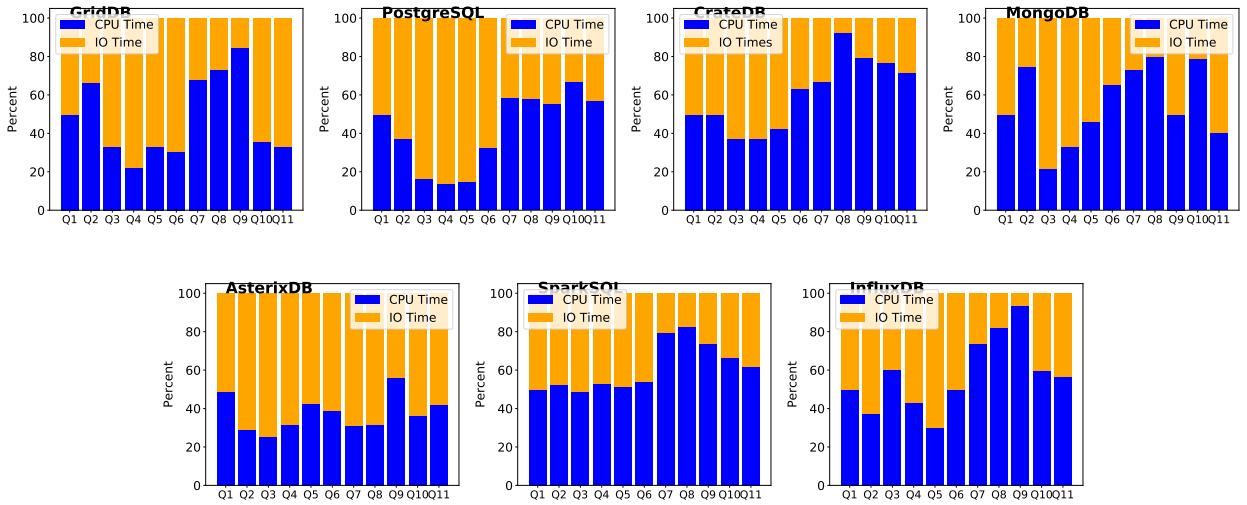


Figure 3.5: CPU and IO breakdown per query for a single node setup.

Metadata Queries Q1 and Q2, both the CPU and IO utilization is very low and balanced across DBMS, since these queries run in few milliseconds. For *Simple Selection and Roll Up* queries Q3-Q6, almost all the systems spent more time on IO operations. CrateDB and SparkSQL spent more time in CPU as they need to decompress and de-serialize after reading data from disk. For *Complex queries* Q7-Q10, AsterixDB spent a higher amount of time on IO than other systems, as it requires scanning comparatively larger records. InfluxDB and GridDB have smaller record size and support compression, hence they spend comparatively less time in IO and have CPU as their bottleneck. Additionally, part of the higher CPU utilization is due to the implementation of unsupported Join and Aggregation operations in the application domain. SparkSQL spends some time doing IO (since it does not support secondary indexes and scans the complete table) but since it needs to perform de-serialization of data [98], 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.

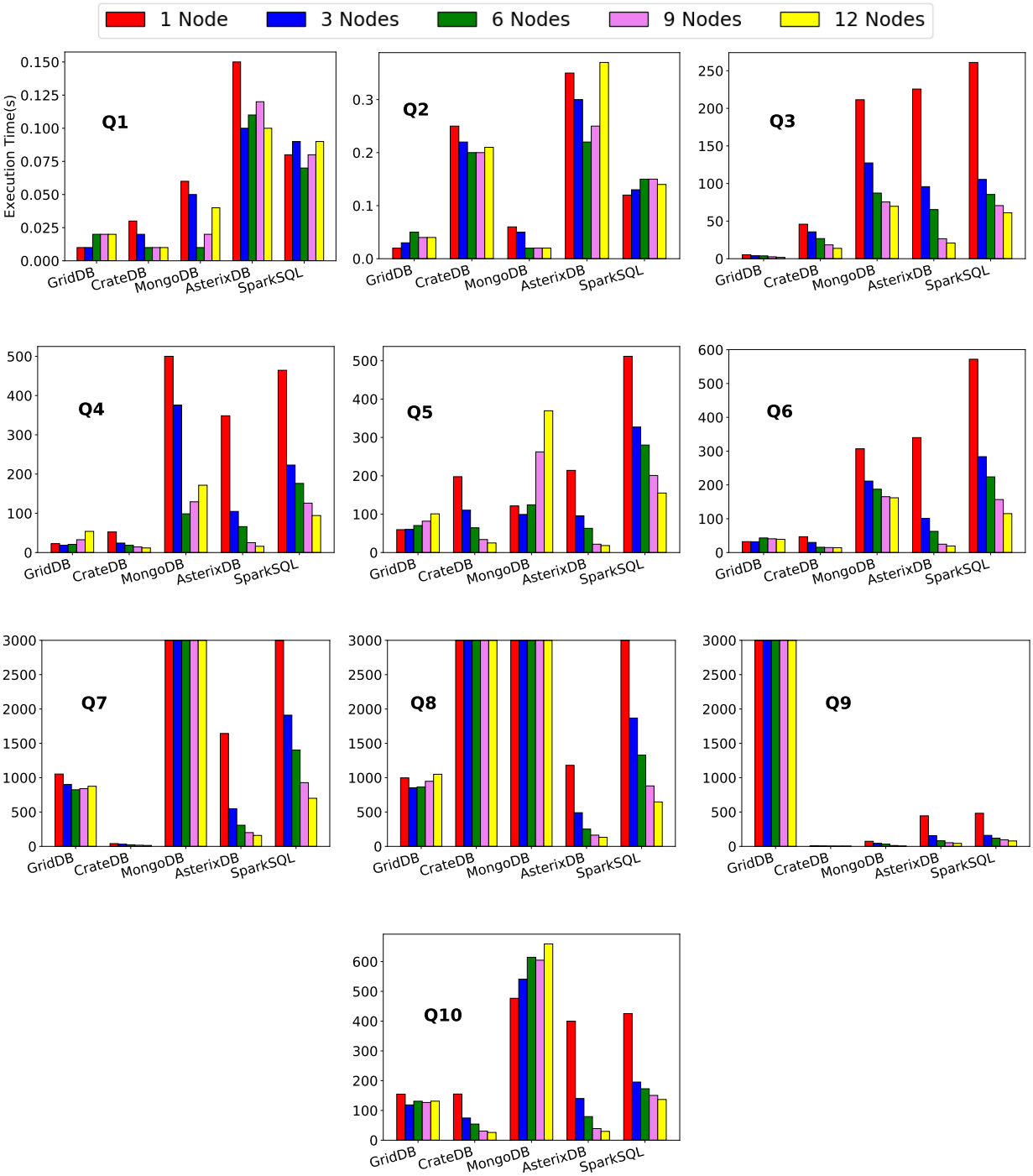


Figure 3.6: Query runtime (s) on small dataset (multi node, 1 hour timeout).

Table 3.5: Ingestion throughput and query latency.

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

3.3.2 Multi-Node Experiments

For multi-node experiments we used larger datasets (see Figure 3.3). Data was partitioned over 1, 3, 6, 9, and 12 nodes (each node is an Intel i5 CPU, 8GB RAM, 800GB HDD, and CentOS 7 machine connected via a 1 Gbps network link). The DBMS instances on each node have the same configuration as in the case of the single node setup. We skipped PostgreSQL and InfluxDB for multinode experiments since the former does not support horizontal sharding natively and the latter supports sharding features only in its enterprise edition which is not open source. For the remaining systems, we chose their most performant mapping for our workload, inferred from the results of the single node experiments. MongoDB, CrateDB, allow data to be partitioned on any arbitrary key, so we partitioned the observation data based on the sensor-id and the semantic observation data on the semantic entity-id. For AsterixDB, data is partitioned on the primary key, as it uses hash-partitioning on the primary key for all datasets. In GridDB, a container is stored fully on a single node since GridDB does not provide an explicit partitioning method. GridDB balances data across nodes by distributing different containers to different nodes based on the hash of their key/name. The information regarding the allocation of containers to nodes is populated to all the nodes in the cluster, making it easy for the client library to locate any container.

Experiment 6 (Insert Performance - Table 3.4): Similar to Experiment 1, GridDB performs well on inserts as expected, although the per tuple insertion time increased w.r.t.

Table 3.6: Query runtimes (s) on large dataset (12 nodes).

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

the single node case even with multiple nodes writing data in parallel; the data size per node has increased by 2.5 times, causing more data flushes from memory to disk compared to the single node case. MongoDB's per tuple insertion time also increased with respect to the single node experiments. Since each tuple is now required to be routed by *mongos* service to the appropriate node based on the sharding key, it was not able to make use of the batched insert, causing its insertion time to increase. AsterixDB's per tuple insert time also increased, but, with this larger dataset, we started seeing the benefits of write optimized LSM-trees as its write performance got considerably better than MongoDB. CrateDB, because of its columnar storage, took the most amount of time in the insert tests.

Experiment 7 (Query Performance - Figure 3.6 & Table 3.6): Figure 3.6 shows the query performance results while Table 3.6 shows the results for a 12 node configuration for the large database with 1.6 billion rows.[4]

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 a 3 node setup. Its performance did not improve with an increasing number of nodes except for query Q3 which requires data to be fetched from only one node.[5] For other queries, that require data to be fetched from multiple containers, GridDB's performance

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

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

does not improve since it natively executes only single container queries and, thus, query processing happens only at a single node. Since the application code we wrote initially to execute queries in GridDB was a sequential program, GridDB was not able to leverage any parallelism from the multi-node setup. We implemented multi-threaded programs to fetch data from multiple containers simultaneously for GridDB. This improved the query performance for GridDB considerably, especially for queries Q3-Q5, as these queries just fetch data from multiple containers based on conditions, without any reduction step due to aggregation type. We did not see much improvement on queries Q7 and Q8.

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

An AsterixDB cluster consists of a single controller node and several worker nodes (storing partitioned data) called node controllers. Applications send queries to the cluster controller, which converts the query into a set of job descriptions. These job descriptions are passed to the node controllers running a parallel data flow execution engine called Hyracks [45]. 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 AsterixDB suffered from the overhead of its job distribution approach.

In a clustered setting, MongoDB uses a router process/node called mongos that fetches

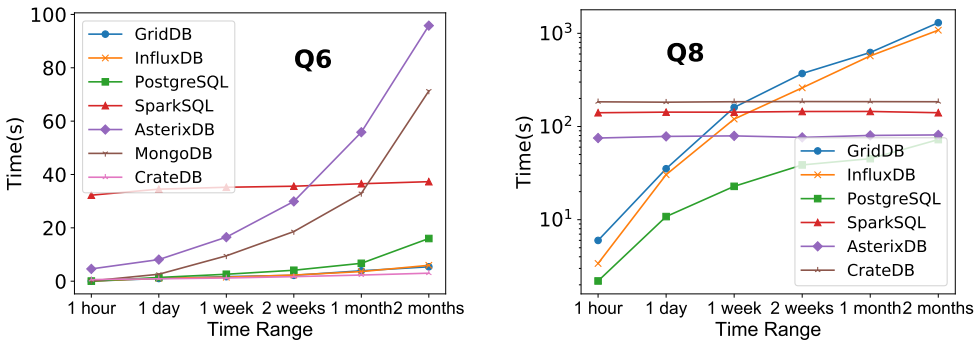


Figure 3.7: Performance with time selectivity.

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

CPU and IO Usage CPU and IO usage for 3 node setup is shown in Figure 3.8. The results (representing the percentage of the total query time spent in CPU and IO on all nodes –master and 3 workers–) are similar in nature to the results for the single node setup. Queries Q1-Q6 are IO bound and Q7-Q10 are CPU bound on most DBMSs.

3.3.3 Mixed Workloads Experiments

We compare system performance under the online mixed workload where queries of the same template are executed in parallel with data ingestion. We used two different levels of data

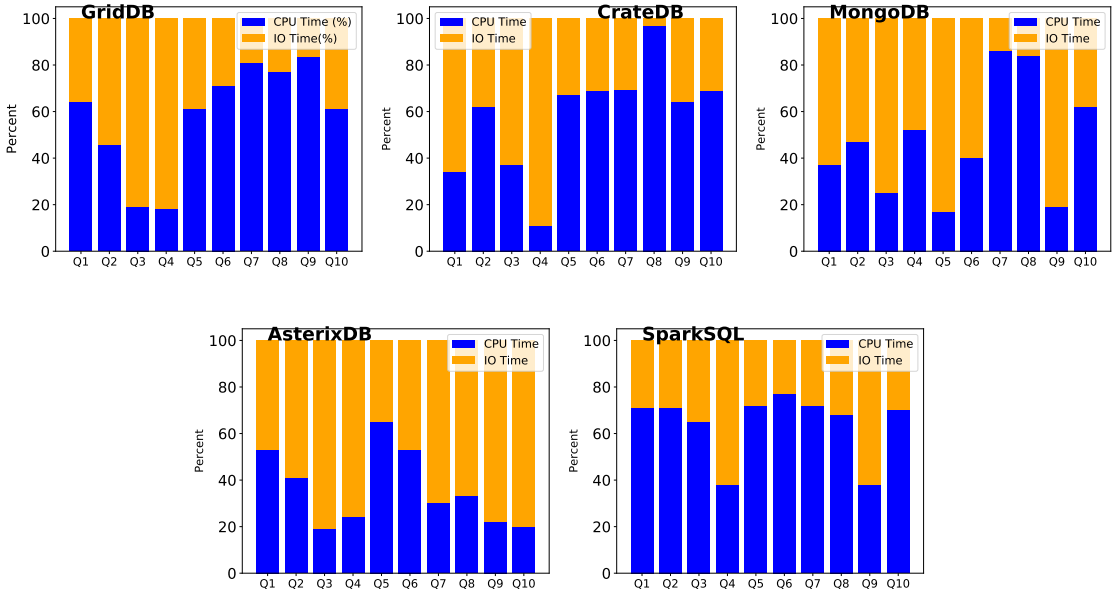
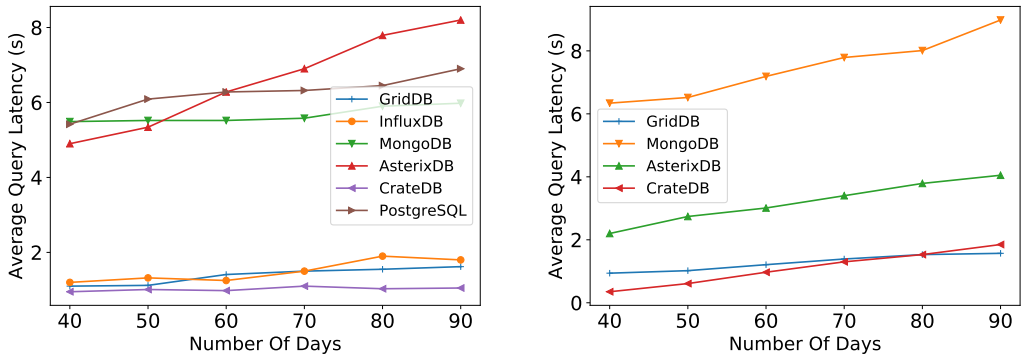


Figure 3.8: CPU and IO breakdown per query for a 3-node setup.

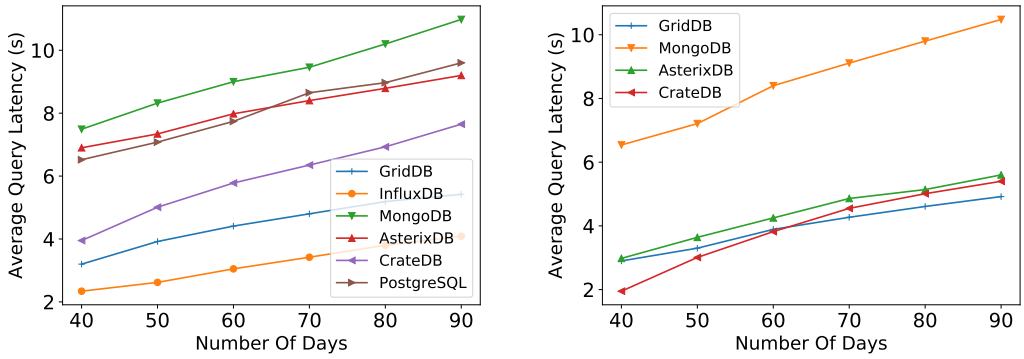
insertion rate, slow and fast, where data is available to insert at the rate of 10,000 and 50,000 observations per second, respectively. The experiment was repeated 6 times with different sizes of data (based on days) already ingested. The experiment starts from a database state where data of a varying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

Experiment 8 (Online inserts and queries - Figure 3.9): Figure 3.9 shows the average query latency for Q6 w.r.t. the number of days for slow data generation rate on a single node as well as a multi node (3 node) setup. The query latency is increasing with increasing number of days for all the database systems, as the data size is increasing. The latency increase rate is comparatively higher in case of faster insert rate as expected since the queries are now running in parallel with a much higher load of inserts. The results for multi-node version of



(a) Single Node (b) Multi Node

Figure 3.9: Q6 in mixed workload (slow insertion rate).

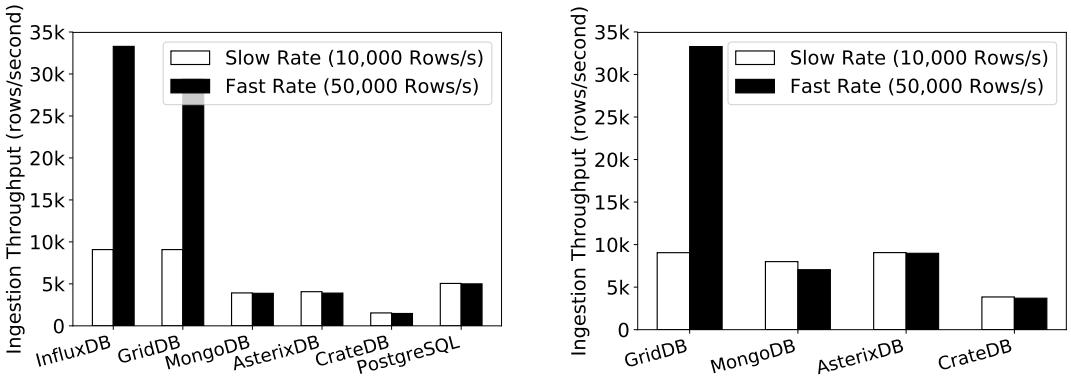


(a) Single Node (b) Multi Node

Figure 3.10: Q6 in mixed workload (fast insertion rate).

the experiment show a similar scalability trend for the DBMSs as discussed in Experiment 7, with AsterixDB, CrateDB showing lower query latency with multi node setup, whereas GridDB and MongoDB did not show much improvement. The same relative performance of the systems is observed for the fast data generation rate is shown in Figure 3.10.

Experiment 9 (Continuous Query - Figure 3.11): We perform an experiment with a mixed workload and the sliding window continuous query (CQ). We set the window length to 10 seconds and the length of the sliding to 5 seconds. Since none of the DBMSs support stream processing, we implemented the continuous query logic on the application side. We buffered occupancy data of all the rooms in the last 10 seconds time window, followed by running a selection query on the DBMS to discover the rooms of type "Lecture Hall" and



(a) Single Node (b) Multi Node

Figure 3.11: Insertion throughput (CQ).

finally joined it with the buffered data. Figure 3.11 shows the results for slow and fast data generation rate on a single node as well as a multi-node (3 node) setup. Since the query is executed as part of the application, the difference in performance can be attributed to the DBMS ingestion rate supported and performance of a simple selection query on a static table. Hence, GridDB and InfluxDB performed significantly better because of their better ingestion performance (discussed in more detail in Experiment 1).

3.4 Lessons Learnt

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

more applicable. Data follows an append-only pattern with rare updates. The volume of data can be very large and hence scale up and scale out functionalities of databases are required.

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

• Specialized timeseries databases are suited for sensor data (fast insertion and time-based selection queries) but they do not provide natural ways to store other data needed to build IoT applications (e.g., spatial relationships, entities, events). For instance, InfluxDB does not provide any way to store non-timeseries data (e.g., metadata). One can overcome such limitations by storing such information in a different database and appropriately co-executing a query across both systems (as we did for InfluxDB using PostgreSQL).

• Document stores are suited to represent heterogeneous data but an embedded representation comes at a high cost in terms of performance compared to a normalized representation. For instance, queries with foreign key based joins, on datasets with normalized documents, run faster compared to queries without foreign key based joins with large denormalized documents. This holds even in a multi-node setting where the smaller collection may not be even present on the same node. Thus, a system that supports document level specification, but (semi)-automatically maps such data to an underlying structured representation could offer the best of both worlds. Examples of such a strategy are closed datasets in AsterixDB and JSON shredding in Teradata [14].

• Time series databases, such as InfluxDB and GridDB, performed well on inserts, simple selection queries, and several complex join queries by exploiting application-level joins. This suggests an opportunity to write wrappers that split SQL queries into a set of queries that can be executed directly on such a system, and continue the remainder of the query execution using application-level operators (e.g., application-level joins). Such a wrapper could provide

a timeseries database with the capability of executing full SQL and still being better in terms of performance in situations where at least one of the tables being joined is small, perhaps, due to selection, as is the case in SmartBench.

• Traditional relational database systems like PostgreSQL do well on both insert and query performance on single node but do not scale horizontally. Document stores, while they scale easily (specifically AsterixDB, which performs very well with a large cluster), have query performance that is not as good as a mature relational system on a single node.

• UDF technologies supported by today's databases are not adequate to enrich data during ingestion. Enrichment, today, is performed outside the database (e.g., in application code, or through a streaming engine) during ingestion. Such an architecture can be sub-optimal specially if complex enrichment function need to run queries to retrieve past data [132]. Co-optimizing enrichment with ingestion (e.g., through batching, or selectively choosing which enrichment to perform in real-time, and which to do progressively, etc.) is an important challenge to support real-time smart applications.

Finally, our key observation (based on the discussion above) is that, like in other domains, while different systems offer different advantages, there is no single system that offers the "best" choice. The emerging field of Polystores [55], which aims to provide integration middleware allowing applications to store different parts of their data in different underlying databases, may be a relevant solution.

Chapter 4

Data Model

In this chapter we describe the TippersDB data model that provides an abstraction to write applications independently of the sensor infrastructure. The data model is a layered data model and consists of four distinct layers as shown in Figure 4.1. The *spatial extent layer* models the physical space or the extent of the smart space (§4.1). The *semantic layer* (§4.2) models entities inhabiting the smart space and their relationships. The *sensor layer* (§4.3) models the type and instances of sensors embedded into the space. The *glue layer* (§4.4) provides mechanisms to translate data from the sensor layer into the semantic layer and vice versa.

4.1 Spatial Extent Layer

TippersDB models the geography in which the smart space is embedded hierarchically in the form of a tree where the root corresponds to the entire extent of the smart space. For instance, it may represent the campus in our running example as root with children including buildings, parks, and walkways which might be further divided into floors in a

building, rooms in a floor, etc. Such a spatial hierarchy naturally supports viewing data and postulating queries at different spatial granularity. For instance, an application may pose an occupancy query at the room, floor, or building level.

Managing spatial data in databases has been extensively studied in the literature [64, 69, 117] with SQL/MM [117] having emerged as a standard to store, retrieve, and process spatial data using SQL. In TippersDB, we adopt SQL/MM, implemented by DBMSs such as SQL Server, DB2, and PostgreSQL, to manage spatial objects. We additionally allow users to define and store hierarchical/topological relationships between spatial objects explicitly. TippersDB supports the following ways to define extents and topological relationships:

```
CREATE T_Extent Name(boundary ST_Rectangle);
CREATE T_TopologicalRel relation(parent Extent, child Extent);
```

where *boundary* is a SQL/MM rectangle which represents the geographical shape of this extent; *relation* represents that the *child* extent is topologically contained inside the *parent* extent.

Through SQL/MM, TippersDB supports a variety of spatial predicates (overlap, intersection, meet, etc.) and spatial operations (intersection, union, area, etc.) over spatial objects. We define a custom version of the difference operator among spatial objects. While SQL/MM supports difference operator between spatial objects (which is a more complex polygon), in determining regions covered by a sensor we will require to partition and represent the resulting difference as a set of rectangles that together represent the space covered by the difference between spatial objects (this requirement will become clear in § 5.2.1).

Difference operator: Given two spatial objects with corresponding bounding boxes A and B, the difference $A - B$, returns the region of rectangle A that does not overlap with rectangle B. Since the resultant region may not be a rectangle, we partition the region into multiple rectangles. Figure 4.2 shows the difference between two rectangles (subtracting a

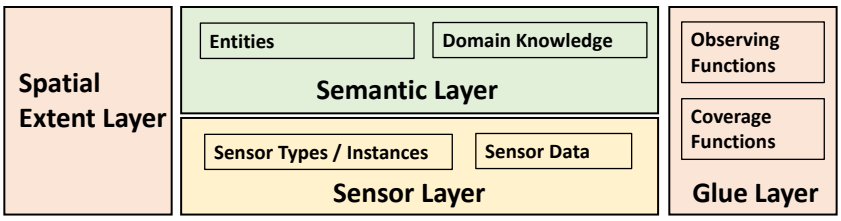


Figure 4.1: TippersDB data model layers.

Algorithm 1: Difference Operation.

Inputs: Two rectangles A and B defined by left bottom $[x_1, y_1]$ and top right $[x_2, y_2]$ co-ordinates
Function ***Difference*(A, B) begin**
  if $Intersect(A, B) \neq \phi$ **then**
    $A_1 = \text{Rectangle}([B.x_1, B.y_2], [min(A.x_2, B.x_2), A.y_2])$
    $A_2 = \text{Rectangle}([A.x_1, A.y_1], [B.x_1, A.y_2])$
    $A_3 = \text{Rectangle}([B.x_1, A.y_1], [min(A.x_2, B.x_2), B.y_2])$
    $A_4 = \text{Rectangle}([B.x_2, A.y_1], [A.x_2, A.y_2])$
  Return the rectangles from (A_1, A_2, A_3, A_4) with area ¿ 0

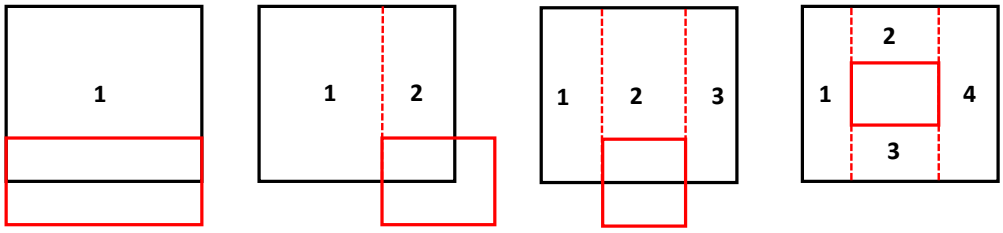


Figure 4.2: Difference operator.

red rectangle from a black one) and the partitioning of the resultant region under different scenarios. Note that for 2-d rectangles, the partitioning can result in at most four rectangles. Also, note that the rectangular representation of the difference between two spaces A and B is not unique. For instance, the partitioning in the second scenario in Figure 4.2 could be done along the x axis instead of the y axis as shown in the figure. To prevent ambiguity, TippersDB gives precedence to partitioning along the y axis over the x axis. Finally, note that the definition (and the algorithm to compute differences) naturally extends to more complex situations where one (or both) operands may be regions defined by a set of rectangles themselves (e.g, as a result of computing a difference between two other rectangles). The extended representation captures spaces such as $((A - A_1) - A_2)....A_n$ where rectangles

$A_1, A_2, ...A_n$ are n rectangles subtracted in sequence from A. The number of rectangles used to represent the different space in this sequence of n subtractions can be shown to be bounded by n^2. [75] considers a more general problem of sequence of differences between d-dimensional spatial objects. We observe that despite the n^2 bound, in practice, such a cover includes much less than n^2 rectangles.

4.2 Semantic Layer

At the semantic level, TippersDB models applications using an extended entity-relation (ER) model that we refer to as *observable entity-relation* (OER) model.

4.2.1 Observable ER Model

OER extends the ER model by defining some attributes and relationships to be *observable*.

Observable attributes. Observable attributes of entities/relationships are those for which changes can be observed (or computed) using data captured by sensors. Figure 4.3 shows an example entity set "occupant" that is either a "visitor" or a "resident". The resident entity set has an observable attribute "vitals" whose value changes with time and can be observed, among others, through a smartwatch or Fitbit. Besides observable attributes, an entity may have additional attributes (e.g, occupant entities have an attribute name) the value of which is not associated with any sensor. We refer to such attributes as *n*on-observable or *r*egular attributes.

Observable relationship. Relationships between entities in an OER model may themselves be observable - that is, can also be observed using sensor data. For instance, in the OER diagram shown in Figure 4.3, *contact* is one such relationship since contact between two

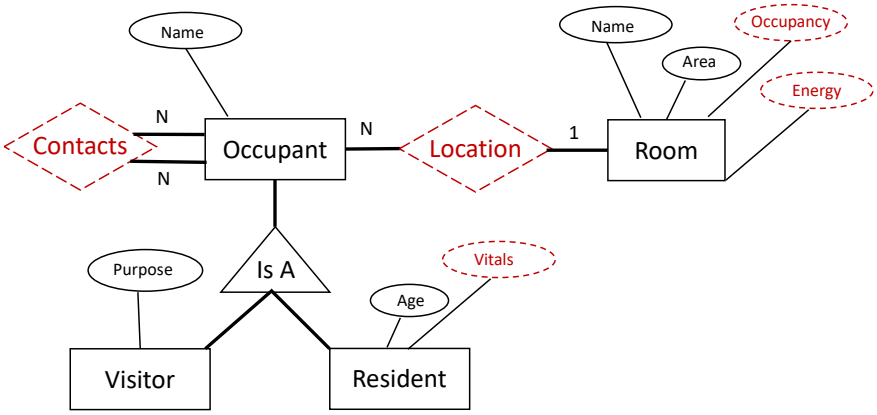


Figure 4.3: Entities with observable attributes/relationships.

occupants can be observed using a Bluetooth sensor in their smartphones (it records all Bluetooth devices in close proximity).

Cardinality Constraints: An observable relationship is characterized as observational 1-1 if an entity e_1 of entity set E_1 at any instance of time can be related to a single entity e_2 of entity set E_2 and likewise any entity e_2 in E_2 at any time is related to a single entity e_1 in E_1. Note that the definition of observational 1-1 is not identical to that in a regular ER model since an entity e_1 can, indeed be related to one or more entities e_2 and e_3 in the entity set E_2. It is just that e_1 cannot be related to both simultaneously. The concept of observational cardinality constraints generalizes naturally to 1-N, N-1, and N-N relationships. For instance, the *L*ocation relationship in Figure 4.3 is an example of an N-1 relationship since a person can be only in a single room at a given time, though a room may have multiple individuals at the same time. In an N-N relationship, entities from both entity sets can be related to multiple entities at a given time instant. The *C*ontact relationship is such an example since multiple people can be in contact with each other simultaneously.

Participation Constraints: An entity set is characterized to have total participation in an observable relationship if every entity of the entity set is related to at least one entity of other entity set at a given time. For instance Occupant entity set in Location relationship is an example of total participation since every person is located in some room at a given

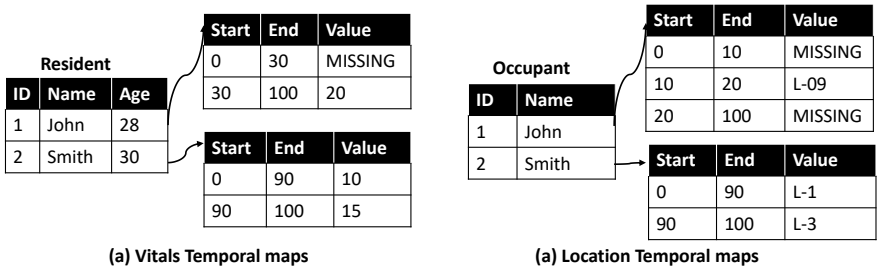


Figure 4.4: Mapping OER to Temporal Maps.

ID	Start	End	Value
1	0	30	MISSING
1	30	100	20
2	0	90	10
2	90	100	15

(a) ResidentVitals

ID	Start	End	Value
1	0	10	MISSING
1	10	20	L-09
1	20	100	MISSING
2	0	90	L-1
2	90	100	L-3

(b) OccupantLocation

ID	Start	End	Value
1	0	60	MISSING
1	60	100	2
1	60	100	3
2	0	100	1
2	0	100	3
3	0	100	MISSING

(c) OccupantContact

Table 4.1: Temporal Relations.

time. Similarly, an entity set is characterized to have partial participation in an observable relationship if not all entities of the entity set are related to an entity of another entity set at a given time. For instance the Room entity set in the Location relationship is an example of partial participation since there can be rooms with no person located in them at a given time.

TippersDB provides the following commands to create entity sets, to add observable properties to the entity sets, and to create observable relationships.

```
CREATE T_ESET Occupant(ID int, name char(20), age int, KEY (ID));
CREATE T_ESET Room(ID int, name char(20), area float, KEY (ID));
ADD T_OBSERVABLE Property vitals TO Occupant (value int));
ADD T_OBSERVABLE Property occupancy TO Room (value int));
CREATE T_OBSERVABLE RELATIONSHIP location (Occupant, Room);
CREATE T_OBSERVABLE RELATIONSHIP Contact (Occupant, Occupant);
```

4.2.2 Mapping OER Model to Relations

Entities and relationships in the OER model are mapped to the relations using the standard ER to relational mapping. However, the observable attributes and relationships cannot be modeled as simple attributes as their values change with time. First, we note that prior literature has explored several ways to represent time-varying data. Broadly, techniques are: *point-based* [125] (that models time as discretized points with a value associated with each time point) or *interval-based* [115] (that models time as a continuous timeline with a value associated with each time interval). TippersDB uses an interval-based representation to represent observable attributes and relationships. Before we describe how observable attributes and relationship sets are mapped to relations, we first describe the concept of *temporal maps*.

Temporal Map. A temporal map (denoted by $\mathcal{T}_{e_i}^{p_j}$) for an entity e_i and a dynamic property p_j is a set of pairs (I, v) where I is a time interval and v is the value of property p_j for entity e_i during time interval I. A dynamic property for an entity has a value at any given time and therefore the time intervals in a temporal map cover the entire time range and are non-overlapping. However, there can be time intervals in a temporal map when the value is `MISSING`.[1] Note that a `MISSING` value is different from `NULL` in SQL. Unlike `NULL`, `MISSING` means that the value exists but has not yet been computed and can be filled in the future; Since temporal maps are defined over the complete timeline, they are associated with a concept of lowest and highest value of time. These values can be set by a user but in the following we assume that the smallest value is 0 and the largest is referred to as *Infinity* (which can be set to an arbitrarily large number). A temporal map can be formally defined as follows:

$$\mathcal{T}_{e_i}^{p_j} = \{(\mathcal{I}_1, v_1], (\mathcal{I}_2, v_2], ...(\mathcal{I}_k, v_k)\} \mid \cup_{j=1}^{k} \mathcal{I}_j = [0, \mathit{Infinity})$$

[1] In TippersDB's layered design (discussed in detail in §5.1), to represent MISSING value, TippersDB reserves a special value for each data type.

Mapping Observable Attributes. To represent an observable attribute of an entity set, TippersDB creates a temporal map for each entity in the entity set and initializes it with a single default time interval of $[0, \mathit{infinity}]$ and a corresponding `MISSING` value. Figure 4.4 shows two temporal maps (one for each entity) for the vitals observable attribute of the resident entity set. Observe that no two intervals in the temporal map overlap, and the intervals together cover the entire time range (with the maximum time (infinity) set as 100).

Mapping Observable relationships. A 1-1 observable relationship is mapped as a temporal map created for the entities of either of the two entity sets. A 1-N or N-1 observable relationship is stored as temporal maps created for the entities on the N-side. For example, for the location relationship in Figure 4.3, which is a 1-N relationship between room and occupant entity set, we create temporal maps for each occupant entity. Mapping of an observable N-N relationship is more complex. We map this by creating temporal maps for each entity in both the entity sets. except in the case of a self-referencing symmetric relationship, in which case the two temporal maps will be identical, and hence only one needs to be stored. Note that the value column in temporal maps created for N-N relationships is a *multiset*, since it stores a list of entities a given entity is related to in a given time interval. Note that N-N relationship introduce a constraint that if a temporal map of entity e_1 of entity set E_1 contains an entity e_2 of entity set E_2 in its multiset value for time interval I, then the temporal map for e_2 must contain e_1 in its multiset value for time interval I. For example, for the contact relationship in Figure 4.3, which is a self-referencing symmetric N-N relationship of the occupant entity set, we create temporal maps for each occupant where the value is a multiset containing all the other occupants that he/she came in contact with at a given time.

The temporal maps corresponding to an observable attribute (or a relationship) associated with each entity in the entity set are stored together in a single relation referred to as the ***temporal relation*** for that attribute. A temporal relation $R_i p_j$ for an observable property

p_j of an entity set R_i consists of a set of triples: a reference to the identity of an entity in R_i, a time interval, and a value p_j for that entity and time interval. Table 4.1a shows the `ResidentVitals` temporal relation containing temporal maps for all entities in the resident entity set for the observable attribute vitals. In case the temporal map consisted of a multiset (as in the case of N-N relationships), we flatten the multiset by inserting a triple for each element in the multiset. For example Table 4.1c shows the `OccupantContact` temporal relation with flattened multisets.

4.2.3 SQL with Temporal Relations.

TippersDB allows users to write SQL queries on top of temporal relations, e.g., the following query retrieves John's location in a time interval [10, 15].

```
SELECT * FROM Occupant O, OccupantLocation OL WHERE O.name='John'
 AND OL.id=O.id AND Overlaps([start, end], [10, 50])
```

Initially the temporal relations contain a single time interval of [0, *infinity*) with a `MISSING` value for all entities. `MISSING` values are computed and materialized by translating appropriate sensor data during query execution. For example, John's location in the Occupant-Location temporal relation (Table 4.1b) is `MISSING` for time interval [0, 10], which will be computed during the query execution. Note that computing a MISSING value may add more rows in a temporal relation, e.g., it may happen that John was in room L-1 during interval [0, 5) and was in room L-2 during interval [5, 10).

4.3 Sensor Layer

At the sensor layer, TippersDB provides a way to specify *sensor type*, to associate *observation type*, and to instantiate sensors in the system. As an example, the following first three

commands define two observation types: `ConnectivityData` (i.e., a data type including device and AP mac address), `ImageData` and `VideoData`, and the last two commands define two sensor types: `WiFiAP` that generate `ConnectivityData` and `Camera ImageData` and `VideoData` observation types.

```
CREATE T_ObservationType ConnectivityData(devMac str, APMac str;
CREATE T_ObservationType ImageData(filelocation str);
CREATE T_ObservationType VideoData(filelocation str);

CREATE T_SensorType WiFiAP([ConnectivityData]);
CREATE T_SensorType Camera([ImageData, VideoData]);
```

After defining observation and sensor types, sensors can be instantiated in the system. Sensors in TippersDB are classified as: (1) *space-based* that generate observations in a physical region and are referred to as *c*overing that space (irrespective of the entity they observe) or (2) *entity-based* that generate the observation about a specific entity (irrespective of the location of the entity). An example of the former is a WiFi access point or a fixed camera deployed at a specific location, while a GPS sensor on a phone carried by an individual or any wearable device that provides input about a specific individual is an example of entity-based sensors. Space-based sensors are instantiated using the following command:

```
CREATE T_Sensor Name(type T_SensorType, mobility bool,
  location Temporal<Extent>,physical coverage Temporal<Extent>);
```

In the above, *mobility* denotes if the sensor is static or mobile/dynamic, *location* refers to the sensor's actual location, and *physical coverage* represents the geographic area in which the specific sensor can capture observations. For instance, a camera could be located in a room, while the physical coverage of the camera is the bounding box surrounding its view frustum. In other words, the physical coverage is modeled deterministically and is simply a function of its location. Both the *location* and the *physical coverage* of a sensor can change with time, either because the sensor is mobile or it was moved at some point. We need to preserve

historical information about the location and coverage attribute to answer historical queries, e.g., "which rooms John visited last year." Hence, in TippersDB, location and coverage are modeled as spatio-temporal attributes. *Entity-based* sensors are instantiated using the following command:

```
CREATE T_Sensor Name(type T_SensorType,mobility bool,entity E_SET)
```

In the above, `entity` refers to the entity that this particular instance of sensor covers.

Different sensors can also be bundled together and be part of a single platform. For example, GPS and Accelerometer sensors can be part of a smartphone or a smartwatch. In that case, the location of all sensors belonging to a platform is determined by that of the platform. To add different sensors to a platform, TippersDB provides the following command:

```
CREATE T_Platform Name(mobility bool, location Temporal<Extent>, sensors [
   T_Sensor])
CREATE T_Platform JohnPhone(true, Temporal<locJS>, sensors [GPS1])
```

Both these models focus on sensor configuration and sensor observations.[2] However, they do not deal with type, mobility, and the dynamic coverage of sensors.

4.4 Glue Layer

The glue layer in TippersDB translates sensor data into semantic information at the application level (i.e., values for the observable attributes and relationships of entities). In this layer, users specify wrapper functions, entitled *observing functions*, for sensor data analysis. Users can also specify *Semantic Coverage* functions that help identifying data from which sensors can be used to generate which semantic observations.

[2]This work does not deal with actuators that perform actions (e.g., switching something on/off).

4.4.1 Observing Functions

Observing functions convert sensor data into semantic observations. These functions are invoked at query time to process sensor data based on the user query. Users first specify their sensor analysis code[3] using the following command.

```
ADD Function Image2Occupancy(Image);
```

The above command adds a sensor data processing function named *Image2Occupancy* that computes occupancy from an image.

To enable such functions to be used as observing functions, the user needs to further connect the type of sensor inputs such a function may take and the observing attribute the function can generate. For instance, a user can write the following code to wrap the *Image2Occupancy* as an observing function.

```
CREATE T_ObservingFunction Camera2Occupancy("Image2Occupancy",
T_Temporal<Room.occupancy>, inputType: [Camera]){}
```

The above command creates an observing function named *Camera2Occupancy* that computes values for observable attribute occupancy (of room entity set) using data from a camera as input. Note that an observing function can take input from more than one sensor. Sensors can be of different types, e.g., an observing function that takes data from both WiFi APs and cameras can be added.

4.4.2 Semantic Coverage Functions

We define a concept of *Semantic Coverage* (*Coverage* for short) with spatial sensors. The semantic coverage of a sensor (or a set of sensors) is defined with respect to an observing

[3]Currently TippersDB supports Python and Java-based functions.

function and it denotes the spatial region for which the observing function can compute values of an observable attribute using the sensor. For instance, for a given camera (sensor), the coverage with respect to face recognition (function) is the region where the image from the camera can be used to detect and recognize the person. The camera image may be used for a different purpose (e.g., detecting people) and its coverage with respect to such a function, used, for instance, to determine occupancy of a region might be different compared to its coverage w.r.t. face recognition. Semantic coverage of a sensor is defined as a function of the physical coverage (which is a property of the sensor, see §4.3).

Similar to the physical coverage, the semantic coverage can also change with time for sensors that are mobile or were moved at some point in time. Formally, the Coverage function is defined below:

$$Coverage(f_l, \{S\}, t) \rightarrow \{(p_i, \Gamma_j)\}$$

where f_l is an observing function, $\{S\}$ is a set of sensors, t is the time instant, p_i is the observable property that f_l computes and Γ_j is a spatial region. Note that the type of a sensor $s \in S$ should be one of the input sensor types of f_l. Also, there should be at least one sensor in S for each input sensor type of f_l. Users can specify semantic coverage as a function in TippersDB, however in case it is not specified, TippersDB uses the physical coverage of a sensor as its semantic coverage.

Based on semantic coverage, TippersDB further defines the notion of Coverage^{-1}. Given an observable property of interest (e.g., vitals, occupancy, location), such a function identifies all possible sets of sensors the input from which can be used to observe the property. For instance, consider a room wherein a person can be located using a camera. Let us further assume that the user can also be located within a room through connection events with a specific WiFi access point. In such a case, Coverage^{-1} function returns both the possible sensors. Formally Coverage^{-1} is defined as follows:

$$Coverage^{-1}(p_i, \Gamma, t) = \{(f_l, \{S\}) | \exists \{(p_i, \Gamma_k)\} \in Coverage(f_l, \{S\}, t)$$

such that $\Gamma_k \cap \Gamma \neq \phi\}$

As will become clear, the Coverage^{-1} function is computed, on the fly, when processing any query that contains observable attributes or relationships. TippersDB uses specialized indexing mechanisms to ensure efficient implementation of the function.

Chapter 5

Design and Architecture

In this chapter, we describe the realization of TippersDB's data model on top of an existing database system. We describe the query-driven translation technique of TippersDB. We also discuss a few optimization techniques that reduce the number of redundant translations (translations that do not affect the query results) and therefore reduce the query latency.

5.1 TippersDB Layered Design

The TippersDB model can be realized either by developing a new database system or by layering on top of an existing one. While the former might enable further optimizations, the latter has often been a preferred route for new technologies (among others, [20, 100, 101]). We followed the layered approach since, first, this approach is largely platform-agnostic and it exploits common features available in a large number of modern databases, viz., indexes, stored procedures, and UDFs. Second, a layered implementation allows organizations/implementations already vested in specific database technology to potentially benefit from TippersDB without having to migrate completely to a new system.

We describe how TippersDB schema, data, functions, and queries are mapped/layered on the underlying database system.

5.1.1 Mapping Schema, Data, and Functions

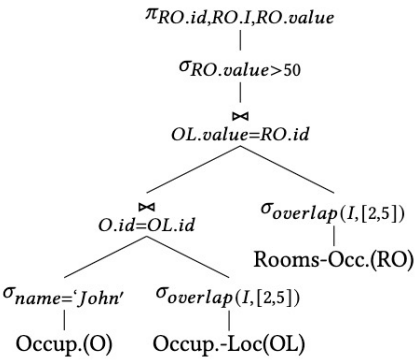
Mapping Space Metadata. TippersDB space model (see §4.1) defines the hierarchical spatial extent including geographical bounds of buildings, regions, and rooms. To store such spatial metadata, TippersDB creates special tables called the `Spatial Metadata` tables in the underlying database.

Mapping Temporal Relations. There are multiple options to map temporal relations/observable attributes to the underlying database: (1) Adding observable attributes to the table created for the entity set; (2) Creating a table for each entity's observable attributes, i.e., a table for each temporal map; (this is similar to the key-container model of [7]) (3) Creating a table for each temporal relation. The first design option will incur a very high space overhead since the space will grow exponentially as the number of observable attributes increases. The second design option can be useful if there are fewer entities and/or most queries fetch data for a single entity at a time. However, in IoT environments, the number of entities can be large and the queries can be ad-hoc, asking for data for multiple entities at the same time. This way, the second option will result in a very large number of tables in most IoT scenarios. Hence, we opted for the third option and create a table for each temporal relation.

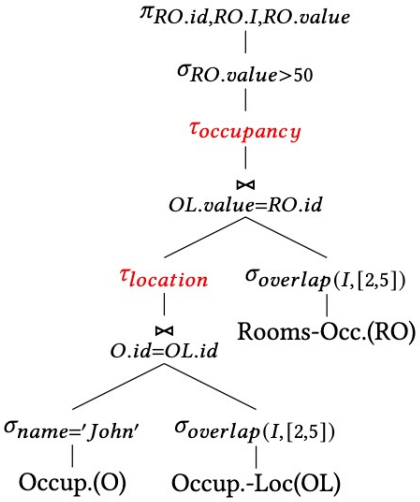
Mapping Sensors and Sensor Data. Static information related to sensors (i.e., sensor types, observation types, mobility) is stored in `Sensor Metadata` tables. Potentially dynamic information related to sensors (i.e., location and coverage area) is modeled as temporal relations and mapped to the database as explained above (i.e., a table per sensor type). Observations from all sensors generating the same type of data (i.e., same observation type)

```
SELECT * FROM Occupant AS O,
    OccupantLocation AS OL,
    RoomOccupancy AS RO
WHERE OL.id=O.id AND
    ((O.name=John AND Overlaps
    ([OL.start, OL.end], [2, 5])) AND
    RO.id = OL.value AND RO.value > 50
```

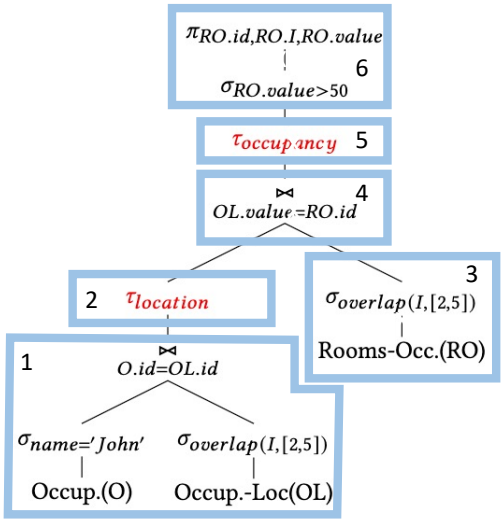
(a) Query on temporal relations.



(b) Original query tree.



(c) Query tree with translation operators.



(d) Query execution in blocks.

Figure 5.1: Query processing in TippersDB.

are stored together in one table. In our reference implementation (see [15]), to support very high intermittent data rates, observations are first pushed to a message queue before they are stored in the database system.

Mapping Functions. TippersDB currently supports Python and Java-based observing and coverage functions (described in §4.4.2). TippersDB also provides a library for developers to wrap their existing sensor processing code into observing functions. The metadata related to the observing functions (i.e., input sensor types, observable property that is generated) is also stored in the `Metadata` tables.

5.1.2 Mapping Queries

In TippersDB, application developers pose queries directly on the application-level semantic data (i.e., temporal relations) using the OER model. For example, an application developer, using the OER model shown in Figure 4.3, who is interested in finding out the occupancy of all rooms that 'John' have visited between time interval [2, 5] and had occupancy > 50, could write the query in Figure 5.1a. The query involves temporal relations for the observable properties location of occupants entity set and occupancy of rooms entity set, i.e., `OccupantLocation` (Table 4.1b) and `RoomOccupancy` (Table 5.1a), respectively.

It is possible that parts of temporal relations required to answer the query are not computed yet (i.e., have `MISSING` values). Hence, the query shown in Figure 5.1a with the corresponding query tree shown in Figure 5.1b cannot be executed directly on the underlying database. To fill the missing values during query execution, TippersDB extends the set of relational operators with a new operator called the ***translation operator*** denoted by τ_{p_j}. The translation operator τ_{p_j} maps `MISSING` values in the temporal relation for observable property p_j to sensor data and observing functions using the spatial information, sensor metadata, and coverage functions. The logic and layered implementation of the translation operator will be explained in §5.2.1 and §5.1.2. TippersDB updates the original query plan by placing translation operators in the query tree, so that `MISSING` values of the temporal relations that are required to answer the query are filled during query execution. Figure 5.1c shows one possible query tree after placing translation operators $\tau_{location}$ and $\tau_{occupancy}$ in the original query tree as shown in Figure 5.1b.

Translation Operator Placement

There are multiple possible positions in the query tree to place translation operators. While placing the translation operators, TippersDB needs to make sure that in the new query plan, no operator (except a *translation operator*) ever sees a `MISSING` value. Hence, a TippersDB

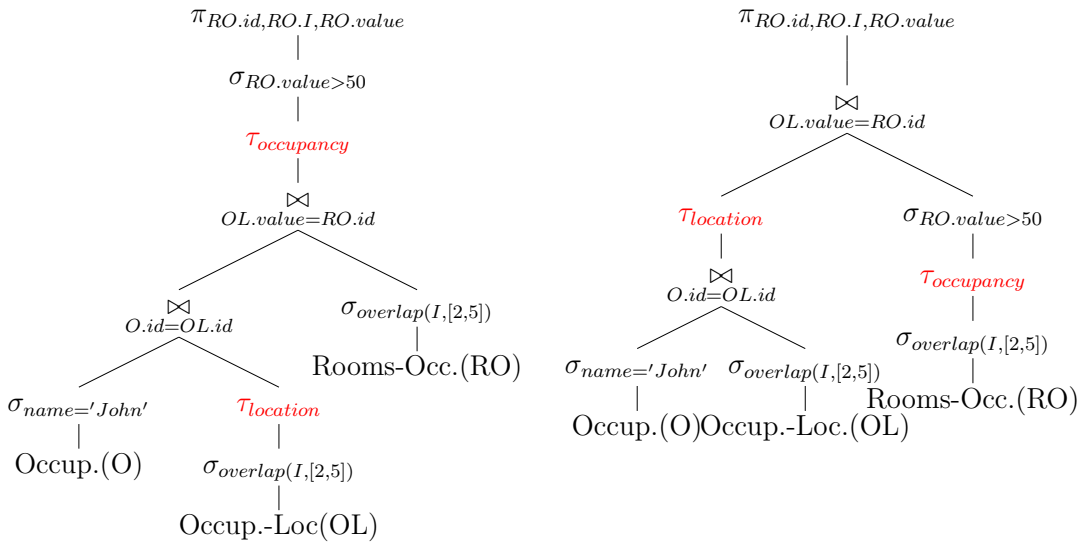


Figure 5.2: Multiple valid query plans.

valid query plan is one that for every non-translation operator η_i (e.g., join, selection, union) involving the value column of a temporal relation $R_i p_j$ (corresponding to an observable property p_j) places a translation operator τ_{p_j} at a node downstream of η_i. The total cost of a valid query plan is the sum of the cost of all translation operators and all relational operators, as:

$$Cost(q) = \Sigma Cost(\tau) + \Sigma Cost(\eta)$$

where τ is a translation operator and η is a relational operator. The cost of a relational operator can be estimated using the standard histograms-based methods [105]. However, the cost of the translation operator cannot be estimated by the underlying database system. We discuss the cost estimation of a translation operator in §5.2.1.

Multiple valid query plans (each with a different cost) can be generated for the same query by placing translation operators at different nodes in the query tree. For example, Figure 5.2 shows two different valid query trees generated by TippersDB for query tree of Figure 5.1b. In both the query trees shown in Figure 5.2, there is a $\tau_{location}$ operator downstream of the join operator involving the value column of `OccupantLocation` relation. Also, there is a $\tau_{occupancy}$ operator placed downstream of the projection operator involving the value column

of `RoomOccupancy` relation.

The number of valid query plans increases exponentially with the number of operators involved in the query. TippersDB selects a plan (with appropriately placed translation operators) with minimum total cost using the traditional dynamic programming algorithm of query optimization [49] used by many cost-based optimizers.

Query Execution

The new query plan cannot be directly executed on the underlying database system as the translation operator is not a standard operator. To execute the query plan, TippersDB divides it into query blocks such that a block contains either only translation operators or only relational operators. Figure 5.1d shows five query blocks created for the query plan shown in Figure 5.1c. After creating the query blocks, TippersDB generates code for a stored procedure called *executor*, that executes each block one by one. For instance, the code for the executor stored procedure generated for the query blocks shown in Figure 5.1d is as follows:

```
1. SELECT * FROM Occupant O, OccupantLocation OL WHERE O.id=OL.eid
   And Overlaps([OL.start, OL.end], [2, 5]) INTO Temp1
2. Translator(Temp1, 'location', Temp3)
3. SELECT * FROM Room R, RoomOccupancy RO WHERE R.id=RO.eid
   And  Overlaps([RO.start, RO.end], [2, 5]) INTO Temp2
4. SELECT * FROM Temp2,Temp3 WHERE Temp2.id=Temp3.value INTO Temp4
5. Translator(Temp4, 'occupancy', Temp5)
6. SELECT * FROM Temp5 WHERE Temp5.value > 50 INTO Answer
```

Note that the query blocks without translation operators are simply executed as a query on the underlying database. The output of these queries is stored in temporary tables that are later used by other query blocks. The query blocks containing only translation operators are

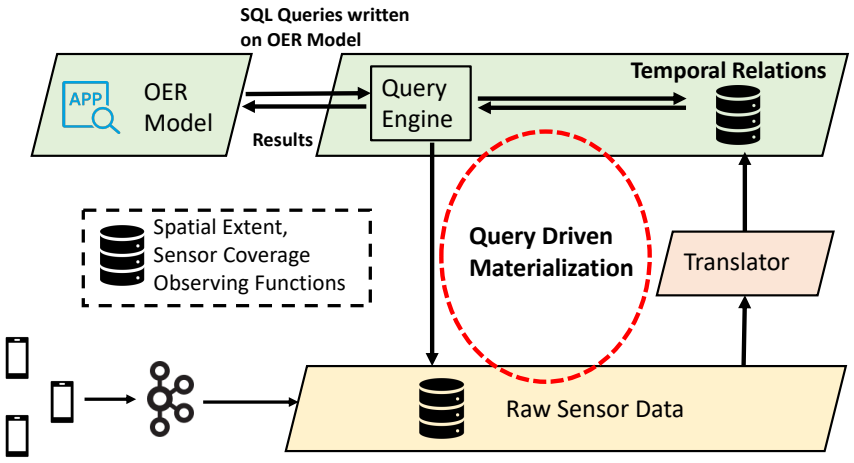


Figure 5.3: Query-driven materialization architecture.

executed using a stored procedure that implements the translation operator[1] and takes a temporary relation (output of downstream query block) as input and fills all missing values for a particular observable attribute in it (§5.2.1 provides details of translation operator). In this query execution strategy, a block cannot be executed unless all its child blocks are completely executed, making it a blocking strategy.

5.1.3 Query Driven Materialization

There are several architectural possibilities to realize TippersDB's layered design. One option is to fully materialize the temporal relations at sensor data ingestion time. This way, queries can directly run on the materialized temporal relations without any query time translation. However, such an architecture can incur a very high ingestion delay. For example, in our running example, analyzing a single WiFi connectivity event takes ∼20ms [83], and analyzing a single camera image takes ∼0.4s. In a medium-sized campus with hundreds of WiFi APs and cameras (producing ∼1,000 WiFi events/sec and ∼100 images/sec), we will need 5 minutes of processing time for locating the person using the data that has been generated in one second, and this processing time is infeasible.

[1]Implementing the translation operator as a UDF is not possible since it adds/updates rows of temporal relations (which is not possible for UDFs). Hence, TippersDB implements the translation operator as a stored procedure.

Another possibility is to not materialize temporal relations at all (i.e., the semantic data is not physically stored and every query is rewritten on top of the raw sensor data to compute the semantic data). This way, every query needs to perform translation from scratch as semantic data computed by previous queries is not stored.

TippersDB takes a *query-driven materialization* (see Figure 5.3) approach where temporal relations are materialized during query execution. Temporal relations are physically stored but can have `MISSING` values denoting which part of the data is not yet materialized and needs to be computed at query time. In this case, queries directly use the already materialized values and only compute the `MISSING` values in the temporal relation. Also, the newly computed values during query execution get materialized into the temporal relations. Note that the current TippersDB implementation performs translation (if needed) only at query time. The approach can, however, be intermingled with techniques to selectively translate sensor data at ingestion or periodically using a background process. Such a generalized approach is an interesting future extension.

5.2 TippersDB Translation

We describe how the translation operator transforms sensor data to compute `MISSING` values (§5.2.1). We discuss strategies to further optimize the translation by exploiting hierarchical data types (§5.3.1) and by indexing temporal relations for interval search queries (§5.3.3).

5.2.1 Translation Operator

The translation operator τ_{p_j}, for an observable property p_j, takes as input a time interval I, a set of entities $\{e\}$ for which the values of p_j is missing and a set of regions $\{\Gamma\}$ that need to be covered to generate the value of p_j to meet the query's requirement. The translation

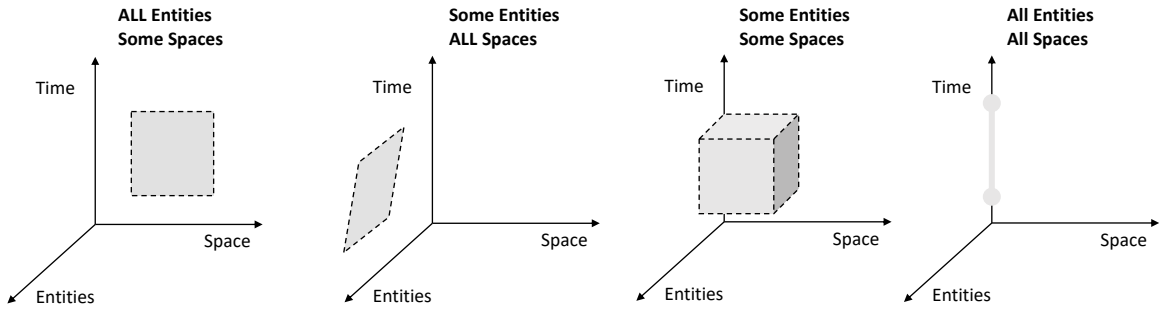


Figure 5.4: Categories of translation operator call.

operator generates and executes a translation plan, $plan(\tau_{p_j})$, that fills all the missing values of p_j for all the entities in $\{e\}$ for the interval I.

Query Cases: The calls to the translation operator can be divided into four categories as shown in Figure 5.4. The entities input to τ_{p_j} could be a set of entities, or it could be ALL, representing the entire set of entities in the corresponding temporal relation. Likewise, the spatial regions in the input to translate could be ALL referring to any region in the extent.

Example 5.1. *Consider the* `RoomOccupancy` *temporal relation (for occupancy property) as shown in Table 5.1(a). Consider also that for a given query, we need to fill the* `MISSING` *values in the temporal relation for tuples corresponding to rooms with id 1 and 2 for time interval [0, 50]. Table 5.1(b) shows a sample translation plan generated by the translation operator, viz.,* $\tau_{occupancy}$*([0,50], {room1, room2}, {room1, room2}). Note that, in RoomOccupancy, the entity itself represents a spatial region, therefore the set of regions to be covered, in this case, is also room 1, room 2. The translation plan shows that the value of occupancy for entity room 1 in interval [0, 10] can be computed through* `CamFunction1` *using data from sensor* `Cam2`*. Likewise, it shows plans to capture occupancy for room 2 for the interval [0,30] and for intervals [20, 35] and [40, 50]. Note that the translation plan for an entity can involve different combinations of functions and sensors for different time intervals if this is deemed as the best plan by TippersDB. To fetch occupancy of all rooms in interval [0, 50] the translation operator can be invoked as* $\tau_{occupancy}$*([0,50], ALL, ALL). This invocation will fill the missing occupancy values for all the rooms (i.e., rooms 1, 2, and 3).*

Algorithm 2: Translation plan generation.

Inputs: $p_j, \{e\}, I, \{\Gamma\}$. // Γ: A region that needs to be covered.
Outputs: Translation plan T.
Function *GeneratePlan*() begin
    T = ϕ
    foreach $\Delta \in I$ **do**
        $T_{entity} \leftarrow EntityBasedPlan(\{e\}, \Delta)$ // if {e} is not ALL
        $T_{region} \leftarrow RegionBasedPlan(\Gamma, \Delta)$
        if $Cost(T_{entity}) < Cost(T_{all})$ **then** T.add(Δ, T_{entity}))
        else T.add(Δ, T_{region})
    Return T
Function *EntityBasedPlan*($\{e\}, \Delta$) begin
    Plan = ϕ
    foreach $e_i \in List\{e\}$ **do**
        Plan.$add(GetBestSensor(e_i, \Delta))$
    Return Plan
Function *RegionBasedPlan*($\{\Gamma\}, \Delta$) begin
    Plan = ϕ
    foreach $\Gamma_i \in \{\Gamma\}$ **do**
        PlanSpace$_i \leftarrow Coverage^{-1}(p_j, \Gamma_i, t)$ // t is a time point in Δ
        $uncovered = \{\Gamma_i\}$, Plan$_i = \phi$
        while $uncovered \neq \phi$ **do**
            Choose $f, S \in$ PlanSpace$_i$ such that rank is lowest
            Plan$_i$.add($\langle f, S \rangle$)
            forall $S' \in PlanSpace_i$ **do**
                if $S'.rank \neq NULL$ **then** $Adjust(\Gamma_{S'}, \Gamma_S, uncovered, S')$
            $uncovered = Difference(uncovered, \Gamma_S)$ // Described in Section 3.1
        Plan.add(Plan$_i$)
    Return Plan
Function *Adjust*($\Gamma_{S'}, \Gamma_S, uncovered, S'$) begin
    $\Gamma'_{S'} = Intersect(\Gamma_{S'}, \Gamma_S)$
    // Intersection with each region $\in uncovered$
    $\{\Gamma''_{S'}\} = IntersectMulti(\Gamma'_{S'}, uncovered)$
    if $\{\Gamma''_{S'}\} = \phi$ **then** $S'.rank =$ NULL
    else $S'.area = S'.area - area(\{\Gamma''_{S'}\})$, $S'.rank = S'.cost/S'.area$

Algorithm 2 shows the logic of the translation operator. Since the coverage of sensors can change with time, TippersDB divides the time interval I into smaller equi-sized sub-intervals of size Δ and generates an optimal sub-plan for each Δ. Δ is chosen to be small enough such that the coverage of sensors is expected to be stable (not changing) in its duration.[2] For

[2]If the coverage changes (e.g., a sensor becomes available/unavailable during a Δ interval), then TippersDB may select a non-

each Δ, TippersDB generates two types of plans (Lines 3-8): entity-based and region-based plans; and of the plans generated, it selects the one with a lowest cost.

Entity-Based Plan (Lines 11-15)

Recall that entity-based sensors always observe a particular entity irrespective of the space they are in. To generate an entity-based plan, TippersDB, for each entity in the input to the translate operator, finds the minimum cost observing function and the entity-based sensor that can observe the given entity for the entire Δ. We generate this plan only when the entities are explicitly listed on the input, i.e., not ALL, or if the region-based plan is not feasible. Note that we could generate an entity-based plan for situations when the input contains ALL, but since number of entities can be arbitrarily large, such plans would be expensive.

Region-Based Plan (Lines 16-28)

TippersDB generates plans using space-based sensors, i.e., sensors that cover all entities in a particular region of the space. Here, TippersDB generates a plan for each region in the queried set of regions. To do so, for each region Γ_i in $\{\Gamma\}$, a plan space that includes all possible plans is generated from which a minimum cost plan will be selected.

Plan Space: To generate the plan space, TippersDB calls the Coverage^{-1} function (discussed in §4.4.2) on the space Γ_i and a time point t from time interval Δ .[3] Recall that the Coverage^{-1} function returns a set of pairs having the observing function f_l and a set of sensors S. Consider Γ_S as the sub-region of Γ_i that can be covered by the set of sensors S using observing function f_l in time interval Δ. Note that different sets of sensors may cover

optimal plan; e.g., a better plan might be possible by dividing Δ into smaller values and choosing different plans for the two different parts of the Δ interval.

[3]We could run Coverage^{-1} for any point of time Δ since Δ is small enough such that the coverage of sensors does not change for any time point in its duration.

ID	Start	End	Value
1	0	10	MISSING
1	10	100	25
2	0	35	MISSING
2	35	40	50
2	40	100	MISSING
3	0	100	MISSING

(a) RoomOccupancy

Entity ID	Start	End	Observing Function	Sensors
1	0	10	CamFunction1	Cam2
2	0	30	WiFiFunction1	WiFi30
2	30	35	CamFunction2	Cam2,Cam3
2	40	50	CamFunction1	Cam3

(b) Translation Plan

Table 5.1: Translation Plan.

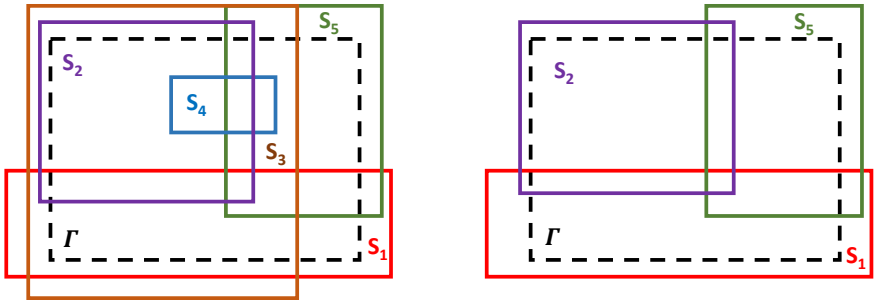


Figure 5.5: Region-based plan generation.

overlapping sub-regions inside Γ_i. For example, Figure 5.5a represents one such plan space and shows a region Γ with different sets of sensors covering different overlapping sub-regions of Γ.

Plan Selection: Executing all the observing functions (with the corresponding sensors) included in the plan space will result in redundant translation work, since multiple sets of sensors might cover overlapping sub-regions of the given region. TippersDB selects a minimum cost subset of the plan space such that the subset covers the entire region. This subset is called a *translation plan*. The condition to select a translation plan is formally defined as:

$$\arg\min_{plan} \sum_{(f_l,S)\in plan} Cost(f_l(S)) \mid \bigcup_{(f_l,S)\in Plan} Coverage(f_l, S, t) \supseteq \Gamma_i$$

where $Cost(f_l(S))$ denotes the estimated amount of time spent in executing f_l on data generated by sensors in S for the time interval Δ and t is any time point in Δ.

The problem of finding the minimum cost plan that covers a region of interest is related to the problem of covering a polygon with minimum number of rectangles which is NP-complete [119]. This polygon covering problem can be easily reduced to the problem of covering a rectangle R by the fewest given rectangles. Let us consider R be a bounding box of a polygon P, and construct a set of rectangles to be used to cover R that include every maximal rectangle contained in P and a set of rectangles obtained from $R - P$ by slicing it into rectangles (so that there is only one way to cover $R - P$). Therefore the problem of covering a rectangle by the fewest given rectangles is also NP-complete.

Thus, to generate the minimum cost translation plan we use a greedy algorithm. First, we sort the plan space based on the ratio of the cost of the function and the area of the sub-region covered, i.e., $Cost(f_l(S))/area(\Gamma_S)$ (denoted as *the rank* of S). We select the entry, (f_l, S), with the lowest rank from the plan space and add it to the translation plan. Note that selecting S will reduce the benefit (i.e., will increase the rank) of other sensors S' that are covering regions overlapping with the region covered by S, since parts of the regions covered by S', are now covered by S. Therefore, we adjust the rank of all other S' in the plan space as $Cost(f_l(S'))/(area(\Gamma'_S) - area(\Gamma''_S))$ where Γ''_S is the region of S' overlapping with S. Next, we remove the region covered by S, Γ_S, from those regions of Γ_i which are not already covered by current sets of sensors in the translation plan. We maintain such regions of Γ_i as a set of regions called $\Gamma_{uncovered}$ (initially containing the entire Γ_i). To remove a covered region Γ_S from Γ_i, we simply subtract Γ_S from $\Gamma_{uncovered}$ (using the difference function mentioned in §4.1). We keep on iterating the above-mentioned steps until the entire region Γ_i is covered or there are no more entries left in the plan space.

Cost estimation for translation operator placement: The previous section describes how translation is implemented during query execution. Recall that before the execution of the query, we need to place translation operators in the query tree (see Section 5.1.2). As discussed there, we need to determine the cost of translation operator when placed at

different nodes in the query tree. Note that since the translation operator placement is done before query execution, we need to develop ways to estimate the cost of a translation operator.

To estimate the cost of a translator operator, TippersDB uses the the cardinality estimates provided by the database. From the input cardinalities, TippersDB estimates the number of entities which will have MISSING values and the estimated number of regions that will need to be covered. The cost of per entity plan of a translation operator will be minimum if the minimum cost observing function has a sensor available for each entity during a time interval. Similarly, the cost of per region plan of a translation operator will be minimum if the minimum cost observing function has a set of sensors available that cover each region. The same idea can be extended to estimate the maximum cost of a per entity and per region translation plan. The minimum and maximum cost of a translation plan is as follows:

$$Cost_{min}(T) = min[N \times Cost(f^{e}_{min}), M \times Cost(f^{\Gamma}_{min})] \times (I/\Delta)$$
$$Cost_{max}(T) = max[N \times Cost(f^{e}_{max}), M \times Cost(f^{\Gamma}_{max})] \times (I/\Delta)$$

where N and M are the estimated number of entities and spaces respectively, f^{e}_{min} and f^{e}_{max} are minimum and maximum cost entity-based observing functions, and, f^{Γ}_{min} and f^{Γ}_{min} are minimum and maximum cost space-based observing functions. We assume that the cost of a translation plan is uniformly distributed and therefore we estimate the cost of translator operator as the average of the minimum and maximum cost. This estimated cost is used to place translation operators appropriately in the query tree (§5.1.2).

5.3 Translation Optimizations

5.3.1 Optimizing Hierarchical Data Types

An entity or an observable property in TippersDB can have a hierarchical data type, for example, as mentioned in §4.1, the location property of occupant relation can be represented at different granularity (e.g., room-level, region-level, building-level). For some hierarchical data types it is possible that the translation at a higher/coarser level level of granularity is less expensive compared to translation at a lower/finer level. For example, LOCATER [83] has shown that locating a person at region-level using WiFi events is much faster than locating a person at the room-level. TippersDB exploits this difference in cost of translation at different granularity of hierarchical data types to reduce the cost of translation by generating more efficient query plans.

Example 5.2. *Consider the query given in Figure 5.1b that finds out the occupancy of all rooms that 'John' has visited between time interval [2, 5] and had occupancy* > 50*. Let us assume, localizing a person at the granularity level of a room or of a region takes 100ms and 10ms, respectively, and finding occupancy of a room takes 50ms. There are 10 regions with 10 rooms per region and the occupancy of rooms is uniformly distributed between 0 to 100. Also, let us assume that John has visited five different rooms belonging to two different regions during the queried time interval. Since the location property is hierarchical, there are the following possible plans (shown in Figure 5.6) for the query of Figure 5.1b.*

- ***Plan 1:*** *first localize John at room-level and then, for the rooms John was in, find the occupancy and check if occupancy > 50. The cost of this plan is 100*100 + 50*5 = 10,250 ms.*
- ***Plan 2:*** *first find out the occupancy of each room and then, for each room where occupancy > 50, check if John was there. The cost of this plan is 50*100 + 100*50 = 10,000 ms.*
- ***Plan 3:*** *first localize John at region-level and then, for the rooms in the regions John was*

*present, check if John was there. For the rooms where John was present, find the occupancy and check if it is more than 50. The plan cost is 10*10 + 100*20 + 50*5 = 2,350 ms.*

- ***Plan 4:*** *first localize John at region-level and then, for the rooms in the regions John was present, find the occupancy. For the rooms where the occupancy was greater than 50, check if John was there. The plan cost is 10*10 + 50*20 + 100*10 = 3,020 ms.*

Observe that plans 3 and 4, that leverage location hierarchy, have much lower translation cost than other plans. Depending on the difference in cost of room/region-level localization and the number of rooms per region, one plan may be better than the other.

In general, an expression $\boldsymbol{p_j}$ `op` $\boldsymbol{a_m}$, where p_j is a hierarchical observable attribute, `op` is one of the comparison operators (e.g., $=, IN$), and a_m is a possible value of p_j, can be transformed to a condition $\boldsymbol{(parent(p_j)}$ `op` $\boldsymbol{parent(a_m))\ AND\ (p_j}$ `op` $\boldsymbol{a_m)}$, where $parent(p_j)$ is the parent node of p_j. Note that the above transformation can be applied as long as the following condition holds.

$$p_j \texttt{ op } a_m \equiv true \implies parent(p_j) \texttt{ op } parent(a_m) \equiv true$$

For example, consider that room 'L-1' is contained inside region 'A'. If an equality predicate "room = L-1" is true, it implies "region = A". Note that this condition might not be true for every hierarchical attribute and predicate. For example, the predicate "room $\neq$ L-1" does not imply "region $\neq$ A". However, this condition is always true for the much more common cases of equality and IN predicates.

Note that this transformation is useful if the cost of translation of the new expression is smaller than the cost of translation of the original expression, i.e., $Cost(\tau_{parent(p_j)}) < (1 - \alpha) \times Cost(\tau_{p_j})$, where α is the selectivity of the predicate $parent(p_j)$ `op` $parent(a_m)$.

This approach of adding hierarchical filters can be extended to more than two levels of hierarchy. For example, a building-level localization using GPS data is even cheaper then WiFi based region-level localization. Therefore, Example 5.2 can be extended to include a chain of

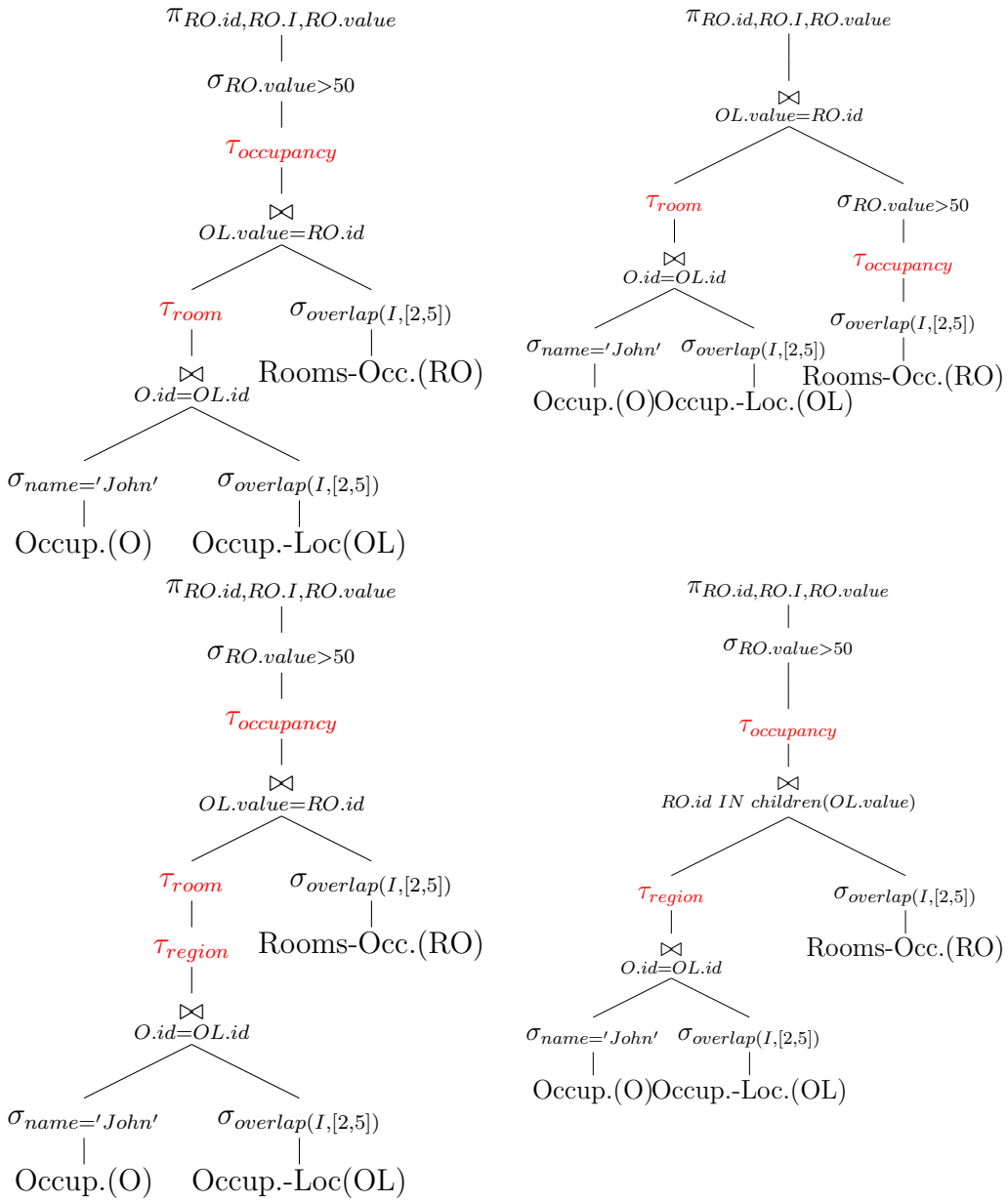


Figure 5.6: Different query plans with hierarchical translation (Plans 1,2,3,4 starting form top-left to bottom-right).

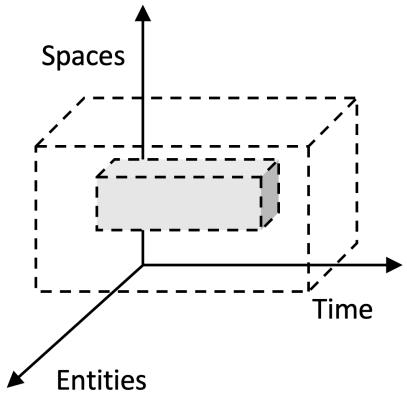


Figure 5.7: Reduced translation space.

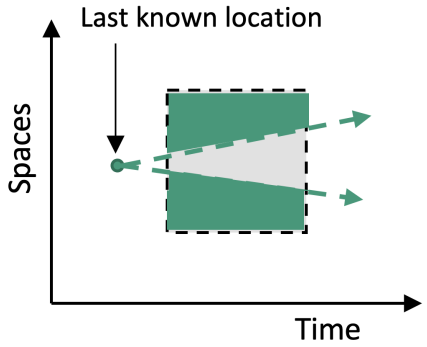


Figure 5.8: Contextual filter based on last known location.

transformations containing first the building-level localization, followed by region-level localization and then finally the room-level localization. This approach of adding transformations based on hierarchy is related to a more general problem of multi-version predicates studied in [81], where multiple cheaper versions of predicates are created for an expensive predicate present in the queries and the less expensive predicates are evaluated first to reduce the number of tuples that are used to evaluate the expensive predicates.

5.3.2 Reducing Plan Space Using Contextual Information

As mentioned in Section 5.2.1 the translation operator for a given observable property p_j and a given time interval Δ, generates a translation plan that covers all the entities or all the spatial regions returned from the upstream operator. This list of entities and/or spatial regions can be very big. However, often, by using some contextual information it is possible to use the set of entities to reduce/prioritize the set of spatial regions to be covered as shown in Figure 5.7. The following example shows how to reduce the set of spatial regions to be covered using contextual information.

Example 5.3. *Consider a query asking for the location of John in a given time interval. In this query the set of entities to be covered is* $\{John\}$ *and set of spaces to be covered is ALL. To answer this query, if a personal sensor for John is not available, TippersDB has to generate*

a space-based plan to cover all the spaces which could be highly inefficient. However, using the fact that a person can physically move only a certain distance in a given time interval, we can limit the spactial regions to be covered to only those spaces that 'John' could have possibly physically moved from his last known location as shown in Figure 5.8.

Similarly, the set of entities to be covered can be reduced or prioritized using contextual information as shown in the following example.

Example 5.4. *Consider a query asking a list all people who were in Room-2065 at a given time. In this query the set of entities to be covered is ALL and set of spaces to be covered is Room-2065. In this case, if there is no space-based sensor available whose physical coverage overlaps with the physical extent of Room-2065, TippersDB has to generate a entity-based plan to cover all the entities which could also be highly inefficient. However, if we find out last known locations of all the entities we can filter out all those persons who could not have physically moved to Room-2065.*

Users can add their application specific logic of using contextual information by adding *contextual filter* function to an observable property. A contextual filter function takes as input a set of entities, spatial regions and a time interval, and returns a subset of entities and the spatial regions as output. Formally, contextual filter functions can be defined as:

$$ContextualFilter(p_j, \{e\}, \{\Gamma\}, I) \rightarrow (\{e'\}, \{\Gamma'\}, I) \text{ such that } \{e'\} \subset \{e\} \text{ And } \{\Gamma'\} \subset \{\Gamma\}$$

where $\{e\}$ is a set of entities, $\{\Gamma\}$ is a set of spatial regions and I is the time interval. During the query execution, before every translation operator τ_{p_j}, TippersDB calls all the contextual filter functions added for the observable property p_j.

Based on output i.e., reduced set of entities and spatial regions, contextual filter functions can be classified into the following two categories.

- **MUST Contextual Filters:** Contextual filter functions that return a subset of

entities and/or spaces such that generating a translation plan for the returned set of entities and spaces is sufficient to answer the query. Therefore, the rest of the entities and spatial regions do not need to be considered for translation. Formally, a contextual filter is classified as a *must* contextual filter if it satisfies the following condition.

$$\tau_{p_j}(\{e\}, \{\Gamma\}, I) \equiv \tau_{p_j}(ContextualFilter(p_j, \{e\}, \{\Gamma\}, I))$$

here τ_{p_j} is the translation operator called for observable property p_j. The context filter described in Example 5.3 is a *must* context filter, as it uses the deterministic physical constraint on the distance moved by a person to reduce the set of spatial regions.

- **MAYBE Contextual Filters:** Contextual filters that return a subset of entities and spatial regions that should be prioritized for translation. However, the remaining set of entities and spaces can not be discarded as the translation of the reduced set of entities and spatial region may not result in generating an answer. For example, consider a query asking for the current location of 'John'. If 'John' has an office, we can first try to locate 'John' in the region containing his office before covering other spatial regions. Similarly, if we know that 'John' is a student, we can prioritize localizing 'John' in lecture halls. However, since it is possible that we are not able to localize 'John' in any of the lecture halls, we may have to consider other spatial regions. Note that using the query execution strategy described in §5.1.2, the *maybe* class of contextual filter functions cannot be implemented since given a set of entities, spatial regions and time interval, the translation operator generates a plan fully covering the entities or the spatial regions. However, in Chapter 6 we will describe a progressive query processing approach, that allows this class of contextual filters to prioritize certain entities and spatial regions to be translated.

TippersDB provides the following command to add a contextual filter function. Note that TippersDB allows multiple contextual filter functions to be added for the same observable property.

```
CREATE T_ContextualFilter UseLastLocation(T_Temporal <Occupancy.location>,
    'MAYBE/MUST')
```

5.3.3 Optimizing Interval Search Queries

TippersDB executes interval-based search i.e., fetches rows from temporal relations having intervals that overlap with a given interval, during query execution. In particular interval based search is used to find coverage of sensors during a particular time interval in implementing the translation operator. Hence, TippersDB needs to have an efficient implementation of such interval search queries to minimize the overall cost of the query. Several indexing structures for interval search queries are proposed in the past [56, 44]. But all these strategies either create or change the indexing structures inside the database system. Since we designed TippersDB using a layered approach, we need to use existing database indexes for interval search queries. One possible way is to maintain separate indexes for both the start-time (referred to as *STI*) and end-time attribute (referred to as *ETI*) of a temporal relation (shown in Table 4.1). To search rows in a temporal relation having time intervals overlapping with a queried time interval $[t^q_{start}, t^q_{end}]$ first, *STI* is used to find out all the records with their start-time less than t^q_{start}. Similarly, *ETI* is used to find out all records with end-time less than t^q_{end}. Finally, an intersection between the *rid*s (i.e., record IDs) of the two lists is performed. This implementation works well as long as the size of the temporal relation is small. However, when the size of the temporal relation grows, the range search with an unbounded side on *STI* and *ETI*, makes the index-based search very inefficient as it may result in a large number of records to be matched before the intersection of *rid* lists even when the length of the queried time interval, $t^q_{end} - t^q_{start}$, is very small.

To avoid scanning the entire temporal relation, we bound the number of records that can satisfy the range-lookup on *STI* and *ETI*. We maintain a parameter called *interval_threshold*

ID	Start	End	Value
1	0	30	MISSING
1	31	60	32
2	0	15	25
2	16	25	MISSING
2	26	50	50

(a)

ID	Start	End	Value
1	0	19	MISSING
1	20	30	MISSING
1	31	50	32
1	51	60	32
2	0	15	25
2	16	25	MISSING
2	26	45	50
2	46	50	50

(b)

Table 5.2: Room occupancy temporal relation with (a) no interval threshold (b) interval threshold of 20.

Sensor	**No.**	**Rows (M)**	**Size (GB)**	**Observing Functions (Cost)**
WiFi	300	80	76	location(room:150ms,region:10ms) occupancy(room:100ms, region:8ms)
WeMo	1,400	37	4	energy(room:20ms)
Hvac	250	15	10	energy(region:50ms)
Camera	1,400	20	840	location(room:200ms); occupancy(room:120ms)
GPS	5,000	50	10	building-location (5ms)
Watch	5,000	50	20	vitals (18ms)

Table 5.3: Sensor Dataset and Functions.

(referred to as ϕ), which represents the maximum length of a time interval in a temporal relation. If the length of an interval is longer than ϕ, then the interval is divided into multiple time intervals, each with a duration less than or equal to ϕ, with each new interval having the same value. Table 5.2(a) shows the `RoomOccupancy` temporal relation without any *interval_threshold* and Table 5.2(b) shows the same temporal relation after setting an *interval_threshold* of 20. This strategy makes the overlapping interval search queries to become bounded. For a queried interval $[t^q_{start}, t^q_{end}]$, we use the *STI* to retrieve records with start-time $\leq t^q_{start}$ and $\geq (t^q_{start} - \phi)$. Similarly, the *ETI* is used to retrieve the records with end-time $\geq t^q_{end}$ and $\leq (t^q_{end} + \phi)$. Finally, to find the overlaps, the intersection between these two lists of record IDs are performed.

Q1	Fetch room-location of John during time (t_1, t_2)
Q2	Find all people in room r during time (t_1, t_2)
Q3	Fetch occupancy of room r during time (t_1, t_2)
Q4	Fetch all rooms with occupancy greater than 100 during time (t_1, t_2)
Q5	Fetch all users co-located (same room, same time) with John during (t_1, t_2)
Q6	Retrieve users who went from room r_1 to r_2 during (t_1, t_2)
Q7	Retrieve average time spent by users in different types of rooms during (t_1, t_2)
Q8	Find occupancy of all rooms visited by John during (t_1, t_2) and with occupancy > 50
Q9	Find rooms consuming > 100 energy units with average occupancy < 50 during (t_1, t_2)
Q10	Find all users who were healthy and visited buildings that were visited by unhealthy individuals prior to them between (t_1, t_2)

Table 5.4: Queries.

5.4 Evaluation

We evaluate the following aspects of TippersDB in our experiments: (1) Ease of application development, flexibility, and extensibility provided by TippersDB; (2) Performance gain by TippersDB through its query-driven translation approach compared to translation done at ingestion; (3) Effect of optimization strategies used by TippersDB.

5.4.1 Experimental Setup

For the experiments, we use the smart campus scenario described in §1.1. We consider that the buildings inside the smart campus are instrumented with WiFi APs, cameras, power meters (WeMo devices [17]), and HVAC sensors. Also, we assume that the campus is inhabited by various people and majority of people carry a GPS-enabled smartphone and a smartwatch. As mentioned in §4.2, the OER model in this scenario consists of people and rooms as main entities. In the model, people have location and vitals as observable properties, and rooms have occupancy and energy consumption as their observable properties. The location property is hierarchical (as discussed in §4.1). The granularity of time for the sensor data and time intervals in temporal relation is set to one second.

Datasets. We used the sensor data generation tool provided in [68] to generate synthetic

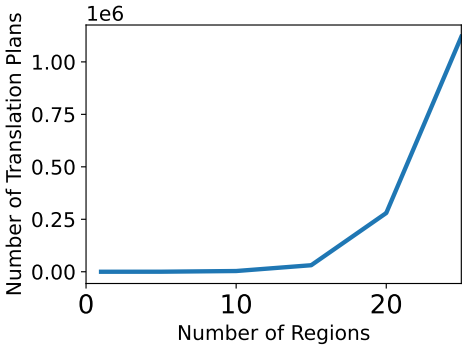


Figure 5.9: Possible translation plans.

EID	Start	End	Observing Function	Sensors
3	50	80	WiFiFunc1	WiFi30
3	80	120	CamFunc1	Cam2
3	120	130	CamFunc1	Cam3
3	130	200	WiFiFunc1	WiFi31

(a) Plan version 1

EID	Start	End	Observing Function	Sensors
3	100	130	WiFiFunc1	WiFi30
3	130	180	WiFiFunc1	Cam2
3	180	210	CamFunc1	Cam3
3	210	250	WiFiFunc1	WiFi30

(b) Plan version 2

Figure 5.10: Sample translation plans.

sensor data for a month for a campus with 25 buildings. Table 5.3 shows the number of instances of each sensor type, the number of rows, and the size of the generated sensor data.

Queries. We selected the following ten queries; see Table 5.4: Queries Q5-Q8 are extracted from SmartBench [68], an IoT database benchmark. All queries are expressed on the semantic model referring to entities and their observable (or not) attributes. Note that during the evaluation we set the value of the time interval parameter, i.e., (t_1, t_2), such that $t_2 - t_1$ is 30 minutes unless explicitly stated.

Observing Functions: We use observing functions to compute building location/occupancy from GPS data, region and room location/occupancy from WiFi data, and room location/occupancy from camera data. Similarly, the energy usage of a room is computed using WeMo data and at region level using HVAC data. We use smartwatch data (i.e., heart rate, O_2 level) to generate a health report of a person. The cost of each function is given in Table 5.3.

5.4.2 Flexibility and Extensibility Evaluation

TippersDB provides a layered data model which acts as a semantic abstraction to developers. This way, it allows them to write applications directly on the semantic/application

data, without handling specific sensors and their produced data. We evaluate the benefit of TippersDB with respect to simplification of smart application development.

Eval 1 - Complexity of Number of Translation Plans: TippersDB creates a translation plan space per query (representing all possible plans), before selecting the best one, thus hiding the complexity of generating and iterating over possible translation plans from the application developer. The number of plans considered by TippersDB can be viewed as an indicator of simplification the system offers - without using TippersDB the developer would need to reason about such plans. Figure 5.9 shows the number of possible translation plans to find room-location of a person in a given set of regions. Observe that to cover a set of 25 regions is more than 1 million possible plans which increase exponentially with the increase in the number of regions to be covered.

Eval 2 - Changing Translation Plans: TippersDB dynamically generates a translation based on the query, available sensors and observing functions. A small change in the query might result in a totally different translation plan. TippersDB hides such complexities from developers who using TippersDB do not have to change the application code to reflect new plans. Consider two versions of query Q3 where the value of (t_1, t_2) in version 1 and 2 is set to $(50, 200)$ and $(100, 250)$, respectively. Figure 5.10(a) and (b) shows the translation plans (generated to fill the missing values in `OccupantLocation` temporal relation) for version 1 and 2, respectively. Observe that, with only a slight change in the query parameter, the translation plans generated are different wrt to the observing function and sensor pairs selected even when the entity of interest (the room) is exactly the same.

Eval 3 - Lines of Code: Developing the localization application mentioned in §1.1 using TippersDB is much simpler (50 lines of code) as compared to writing directly on sensor data. In the latter, sensor data processing functions are explicitly called by the application code which increases its complexity (500 lines of code).

Entity Set	Occupants			Rooms	
Property	location	vitals	contacts	occupancy	energy
Time(days)	70	8	17	34	12

Table 5.5: Exp 1 - Eager translation.

Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Translation	85%	83%	70%	88%	83%	81%	95%	84%	82%	91%
Planning	12%	15%	15%	9%	12%	17%	4%	14%	15%	8%
DB Queries	3%	2%	5%	3%	5%	2%	1%	2%	3%	1%

Table 5.6: Exp 3 - Abstraction overhead.

5.4.3 Performance Evaluation

The following experiments were performed on a server with 16 core 2.50GHz Intel i7 CPU, 64GB RAM, and 1TB SSD. Both the sensor data and the temporal relations were stored in PostgreSQL.

Exp 1 - Eager Translation: We mentioned in § 1 that processing/translating the entire sensor data to generate meaningful observations is not practical. Table 5.5 shows the amount of time required to process all the sensor data at ingest using the functions and their cost mentioned in Table 5.3. For example, Table 5.5 shows that time to process 37M rows of WeMo data using the observing function that takes 20ms per row to compute energy used by a room will be $37 \times 10^6 \times 20\text{ms} \approx 8$ days. Complete translation of GPS, WiFi, and Camera data captured in a month to generate location values would take **≈70 days**, which is highly impractical.

Exp 2 - TippersDB performance and effect of optimizations: Figure 5.11 shows the total execution time of queries in TippersDB including effect of the two optimizations: a) strategy to reduce translation cost for observable properties of hierarchical data types (see §5.3.1), and b) Indexing of the temporal relation using the strategy in § 5.3.3 with *interval_threshold*=15 mins. Figure 5.11 shows that queries Q1, Q4, Q5, and Q8 benefit extensively from the hierarchical optimization - the total query time for Q1 and Q4 reduced

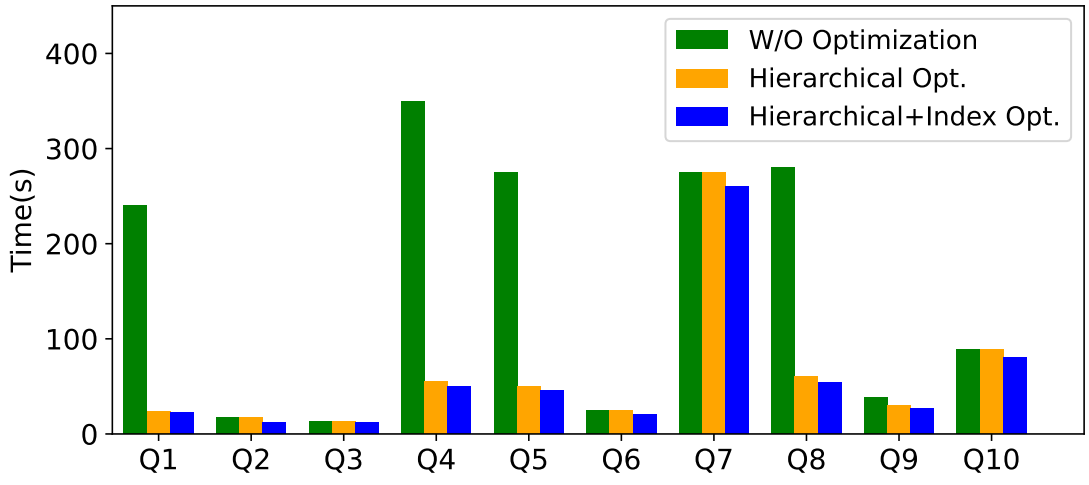


Figure 5.11: Exp 2 - Performance and effect of optimizations.

Query	**Avg. Diff.**	**Query**	**Avg. Diff.**
Q1	4.8%	Q6	10%
Q2	3.9%	Q7	4%
Q3	2%	Q8	4.5%
Q4	8.8%	Q9	6.2%
Q5	3%	Q10	2.5%

Table 5.7: Accuracy of translation operator cost estimation.

from 240s to 23s and from 350s to 46s, respectively. The optimization does not benefit Q10 since it is already at the coarse level. Q2 and Q3 that are room-based queries require that users be localized to fine (room) level and cannot exploit the optimization. Likewise, Q7 requires localizing all users at room-level. In each of the queries, Figure 5.11 shows indexing reduces the execution time by 5 to 10% of the cost. Note that of all the queries Q7 is difficult to optimize since it requires all users to be localized to room level. However, note that the times are still only a small fraction of the eager approach.

Exp 3 - Abstraction Overhead: Table 5.6 shows for each query Q1-Q10, the percentage of the total time of query execution spent in translation, generating translation plans and executing queries on the underlying database(executing a query in TippersDB may result in

multiple queries on the underlying database as mentioned in §5.1.2). For all queries, more than 80% of time is spent in translation and only a small part of the total time is spent in translation planning and running queries on the database.

Exp 4 - Accuracy of Translation Operator Cost Estimation: In this experiment we study accuracy of translation operator cost estimation technique mentioned in §5.1.2. We measure accuracy of the cost estimation technique by running each query 10 times and calculating the average of the difference (show in Table 5.7) of the actual translation cost and the estimated translation cost for each query as a percentage with respect to the actual translation cost. Observe that the difference between the estimation cost and the actual cost is at most 10% of the actual cost.

Exp 5 - Effect of contextual filter functions In this experiment we study the effect of contextual filter functions. For this experiment, we added a contextual filter function for the location observable property that limits the set of spatial regions to be covered to find a person's location using his last known location as describe in Example 5.3. With the help of this contextual filter function the query execution time for Q1 reduced from 230s to 45s as the numbers of spatial regions to be covered reduced from 1400 to 280.

Exp 6 - Effect of Selectivity: In this experiment we study the effect of the query selectivity on the TippersDB query execution time. For this purpose, we vary the selectivity of query Q1 by changing the duration of the time interval i.e., (t_1, t_2) parameter. Figure 5.12 shows that the total query execution time (and hence the translation cost) of Q1 increases linearly with increasing duration of the time interval.

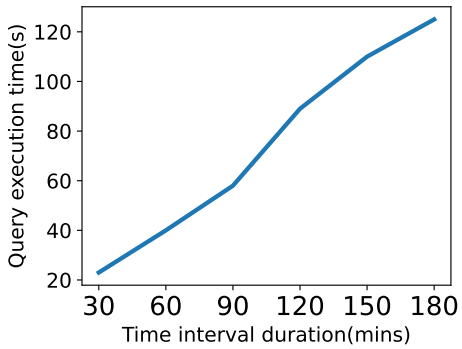


Figure 5.12: Exp 6 - Effect of query selectivity.

5.5 Conclusion

In this chapter we described the realization of TippersDB data model on top of an existing database system. We described the translation mechanism of TippersDB and showed how TippersDB integrates query processing with translation. We also introduced a few optimization techniques that reduce the number of translations to be done. Note that, this chapter describe a blocking approach of query processing where the intermediate query results are fully materialized. This approach can increase the wait time for the user and also can encounter high storage overhead if the intermediate results are large. In the next chapter we will see a modified query processing approach that progressively provides users with answers and also does not require any explicit materialization of intermediate results.

Chapter 6

Progressive Query Processing

In this chapter, we address the issue of increased query latency and user wait time which arises due to query driven translation strategy mentioned in Chapter 5. In this chpater, we describe a query processing approach that progressively returns results to the end users/analysts instead of making them wait for the query to finish executing. In particular, we develop techniques to progressively translate sensor data and incrementally compute answers. We begin by defining the semantics of progressive query processing and then describe ways to achieve progressiveness.

6.1 Progressive Queries

Similar to EnrichDB [61], to return progressive results to users, we first discretize the query execution time into *epochs*: $\{ep_0, ep_1, \ldots, ep_n\}$ as shown in Figure 6.1. In each epoch, TippersDB refines previously produced answers, by adding or retracting tuples from the set of tuples returned in the previous epochs. Specifically, the answer set $Ans(Q, ep_i)$ for a query

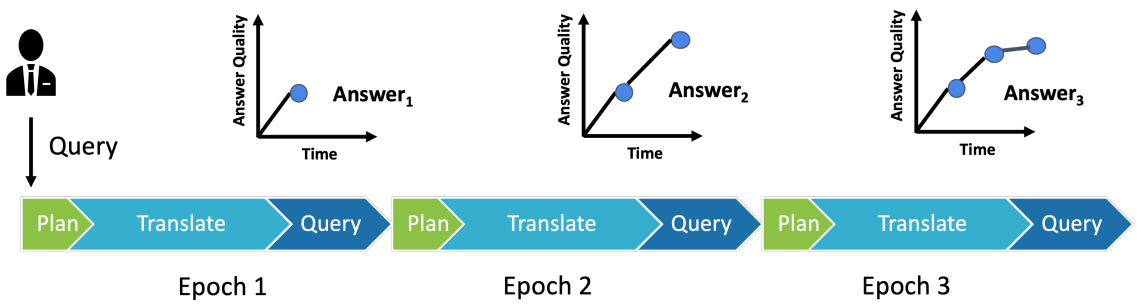


Figure 6.1: Progressive query processing.

Q at the end of an epoch ep_i is defined as follows:

$$Ans(Q, ep_i) = \{Ans(Q, ep_{i-1}) \cup \Delta(Q, ep_i)\} \setminus \nabla(Q, ep_i) \tag{6.1}$$

where $\Delta(Q, ep_i)$ and $(\nabla(Q, ep_i))$ is the set of tuples added to and removed from query answers returned before epoch ep_i.

6.1.1 Progressive Score

The effectiveness of TippersDB progressive query processing is measured using a progressive score (similar to other progressive approaches used in [99, 31]):

$$\mathcal{P}(Ans(Q, ep_n)) = \sum_{i=1}^{n} W(ep_i) \cdot [Qty(Ans(Q, ep_i)) - Qty(Ans(Q, ep_{i-1}))] \tag{6.2}$$

where $\{ep_1, ep_2, \ldots, ep_n\}$ is a set of epochs, $W(ep_i) \in [0, 1]$ is the weight allotted to the epoch ep_i, $W(ep_{i-1}) > W(ep_i))$, Qty is the quality of answers, and $[Qty(Ans(Q, ep_i)) - Qty(Ans(Q, ep_{i-1}))]$ is the improvement in the quality of answers occurred in epoch ep_i. Assigning higher weights to the earlier epochs provides higher importance to the improvement in quality in the earlier epochs. Since weights W_i in the progressive score defined above are decreasing, optimizing the progressive score is equivalent to selecting in every epoch a set of

entities and spaces to translate such that it results in maximum increase in quality in the following epoch, that is, $Maximize(Qty(Ans(Q, e_i)) - Qty(Ans(Q, e_{i-1})))$.

6.1.2 Quality

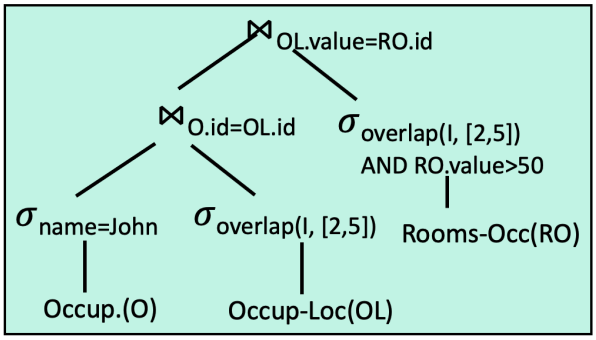
The quality Qty in Equation 6.2, for a set-based query answer corresponds to a set-based quality metrics such as F_1-measure [106] defined as follows.

$$F_1(Ans_w) = \frac{Pre(Ans_w) \cdot Rec(Ans_w)}{(Pre(Ans_w) + Rec(Ans_w))} \tag{6.3}$$

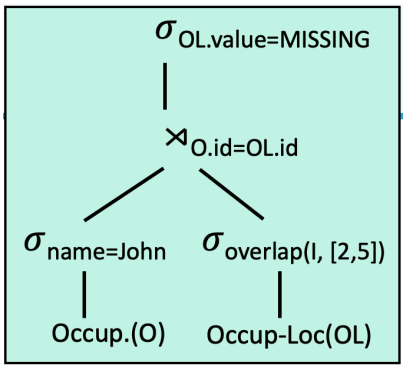
where Ans^{real} is the real answer of the query in ground truth set G, Pre is precision, i.e., $Pre(Ans_w) = |Ans_w \cap Ans^{real}|/|Ans_w|$, and Rec is recall, i.e., $Rec(Ans_w) = |Ans_w \cap Ans^{real}|/|Ans^{real}|$. The quality of an aggregation query could be measured using root-mean-square error [73] or mean-absolute-error [134].

6.2 Probe Queries

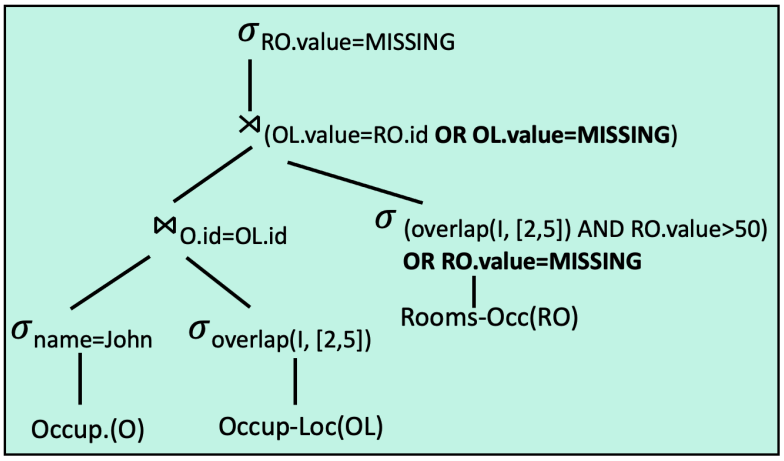
Before the epoch-based progressive query execution, it is important to find out for each observable property p_j involved in the query, the *minimal* set (as small as possible subset) of entities of temporal relation $R_i p_j$ that have MISSING values and may have an impact on the query results. Similarly, for each involved observable property p_j we need to find the minimal set of spaces that need to be covered to answer the query. One could add all the entities that have MISSING values to this set, however, it would result in a significant number of redundant translations, i.e., the tuples that do not satisfy predicates on the regular attributes may be added to this set for translation. Therefore, we generate *entity and space probe queries* for each observable property p_j (that is part of the query) to identify the minimal subset of entities and spaces respectively.



(a) Original Query



(b) Entity probe query for OccupantLocation relation.



(c) Entity probe query for RoomsOccupancy relation.

Figure 6.2: Entity probe query generation.

6.2.1 Entity Probe Query

To compute the minimal subset of entities containing MISSING values, TippersDB generates an entity probe query (denoted as $EPQ(p_j)$) for each p_j that needs to be translated to execute Q. TippersDB uses the following three ideas to generate the probe queries:

- *Selection Conditions on Regular Attributes:* Given a selection condition on a regular (non-observable) attribute, an entity that does not satisfy that condition could be dropped from consideration for translation since the tuples consisting of this entity will anyways be dropped during the query execution.

- *Join Conditions:* Given a join condition on a temporal relation R_ip_j, if for an entity R_ip_j, there is no tuple in R_ip_j that satisfies the join condition, then that entity can also be dropped from the translation process.

- *Prior Translation:* We need to consider only those entities that have MISSING values for the observable property p_j. All the other entities that got translated as a result of previous queries can be discarded.

Example 6.1. *We illustrate how the entity probe queries can exploit the selection and join conditions using the query shown in Figure 6.2a that finds out the occupancy of all rooms that 'John' have visited during time interval [2, 5] and had occupancy* > 50. *In this query, for the temporal relation OccupantLocation, there is a selection condition on the time interval, i.e., the condition* $overlap(I, [2, 5])$*, therefore we need to consider only those rows of OccupantLocation that have time interval overlapping with [2,5]. Also, since we only need to translate entries related to 'John', the rows of OccupantLocation relation can be further reduced by performing a semi-join with the Occupants table as shown in Figure 6.2b. Similarly, Figure 6.2c shows the entity probe query generated by TippersDB to reduce the number of rows of RoomOccupancy relation that need to be translated. The entity probe query for RoomOccupancy exploits the selection condition on the time interval along with the join condition requiring translation of only those rooms that 'John' has visited.*

Following are the steps to generate an entity probe query for an observable property.

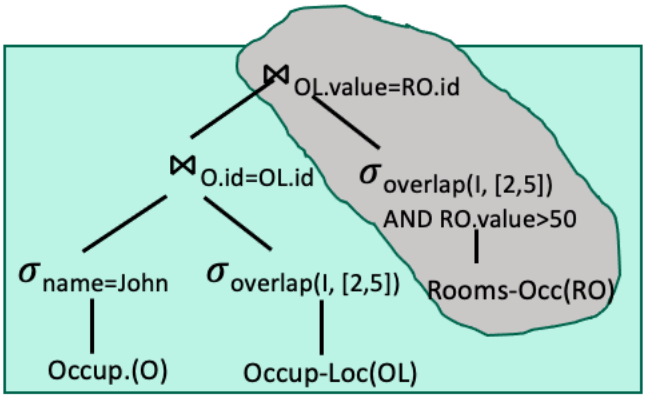
[**Step 1**]: ***Rewrite of Selection Condition (*** $\sigma_C(R_ip_j)$ ***):*** Given a selection condition C of the form $C' \wedge C_k$ where C_k is a predicate on the value column of temporal relation R_ip_j e.g., $R_ip_j.value = 10$. The condition C is rewritten as $(C') \wedge (C_k \vee R_ip_j.value =$MISSING). This step keeps all those tuples that have a MISSING value and are required to be translated to answer the query.

[**Step 2**]: ***Generating Join Graph***: Given a modified query after rewriting it using Step

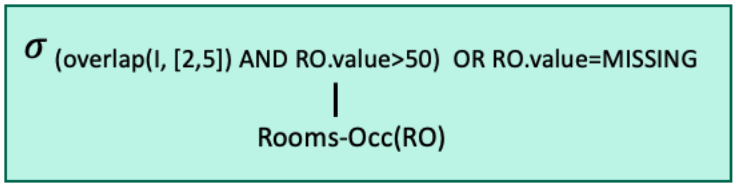
1, we generate a ***join-graph*** from the query. The purpose of the join graph is to find out for a temporal relation R_ip_j in the query which join conditions with other relations can be utilized to reduce the number of tuples of R_ip_j that require translation. In the join graph, the nodes correspond to *reduced* relations, i.e., relations with the selection conditions applied on them. An edge between two nodes shows the join conditions between two relations (based on the original query). Next, from each edge of the join graph, all the join conditions containing a predicate C_k on the value attribute i.e., $R_iP_j.value = R_l.A_1$ is rewritten as $C_k \vee R_iP_j.value$ =MISSING. In a query tree *union*, *set-difference*, or *cross product* operators are ignored, since they cannot be utilized to reduce the number of tuples in *probe queries* apart from the join conditions.

[Step 3]: *Semi-join Program Generation*: Given the join graph as an input, for each node N_i in the graph, this step generates a set of semi-join programs for N_i to reduce the number of tuples of N_i that require translation. For N_i, semi-join programs are generated by exploiting join conditions among nodes of the graph. For node N_i, this step starts from node N_i in the join graph and generates a spanning tree, denoted as $ST(N_i)$, that contains all nodes of the graph with the minimum possible number of edges (using breadth-first traversal). From $ST(N_i)$, multiple semi-join programs are generated based on the join conditions in $ST(N_i)$. Semi-join programs for a node N_i are generated in a bottom-up manner from $ST(N_i)$ starting from the children nodes and reaching up to N_i. For each node encountered in the path, a semi-join program is generated. The nodes in $ST(N_i)$ are traversed in a breadth-first order from the leaf node to the root node. All the semi-join programs between the leaf node and their immediate parent nodes are created first. This step is continued until all the paths from the leaf node to the root node are consumed. This step for semi-join program creation is based on the seminal work on semi-join reduction given in [42].

[Step 4]: *Generating `probe queries`*: Given the semi-join programs (obtained in the previous step), this step simply converts the semi-join programs into queries.



(a) Original query.



(b) Space probe query for OccupantLocation relation.

6.2.2 Space Probe Query

TippersDB generates space probe queries to find the minimal set of spaces that need to be covered to answer a query. Note that the space-probe queries are generated for only those observable properties that are of spatial data types e.g., location property of occupant relation. To generate a space probe query for an observable property p_l (with $R_i.p_l$ as the corresponding temporal relation) having a spatial data type, we exploit the selection and join condition on the value column of temporal relation $R_i.p_l$.

Example 6.2. *Consider the query shown in Figure 6.3a that finds out the occupancy of all rooms that John have visited between time interval [2, 5] and had occupancy* > 50*. To execute this query, we only need to localize John in the rooms that either have an occupancy greater than 50 or their occupancy is MISSING (is not computed yet) in the time interval [2, 5]. Figure 6.3b shows the space probe query generated for the OccupantLocation temporal relation using the observation made above. Note that, for RoomOccupancy temporal relation, the set of entities is the same as the set of spaces, and therefore, there is no separate space*

probe query generated for the RoomOccupancy relation.

TippersDB performs the following two steps to generate the space probe queries for an observable property p_l having a spatial data type.

[**Step 1**]: Similar to the step 1 of entity probe query generation strategy, we first rewrite all the selection conditions C of the form $C' \wedge C_k$ (where C_k is a predicate on the value column of a temporal relation $R_i p_j$) to $(C') \wedge (C_k \vee R_i p_j.value$ =MISSING).

[**Step 2**]: Then we exploit the selection/join condition on the observable property p_l (which is of a spatial data type). If there is a join condition C on the value column of the temporal relation $R_i p_l$, of the form $R_i.p_l.value = R_j.A_1$, then the space probe query for p_l is the sub-query rooted at the node containing the join condition C in the query tree. Note that, if instead of the join condition there is a selection condition on $R_i p_l$ of the form $R_i p_l.value$ IN $(a_1, a_2, .., a_m)$, the space probe query is simply 'select $(a_1, a_2, ..a_m)$' as the only spaces required to be covered are $a_1, a_2, ..a_m$.

6.3 Epoch-based Query Processing

In this section, we describe the epoch-based query execution in TippersDB. Figure 6.4 shows the complete end-to-end pipeline of TippersDB's progressive query processing approach. At the high level, the progressive approach consists of the following steps.

- We compute the minimal set of entities and spaces (for each observation property) to be covered to answer the query.
- We divide the query execution into multiple epochs. In each epoch, we select a subset of entities and spaces to be translated.
- We create a translation plan for the selected entities and spaces. The translation plan

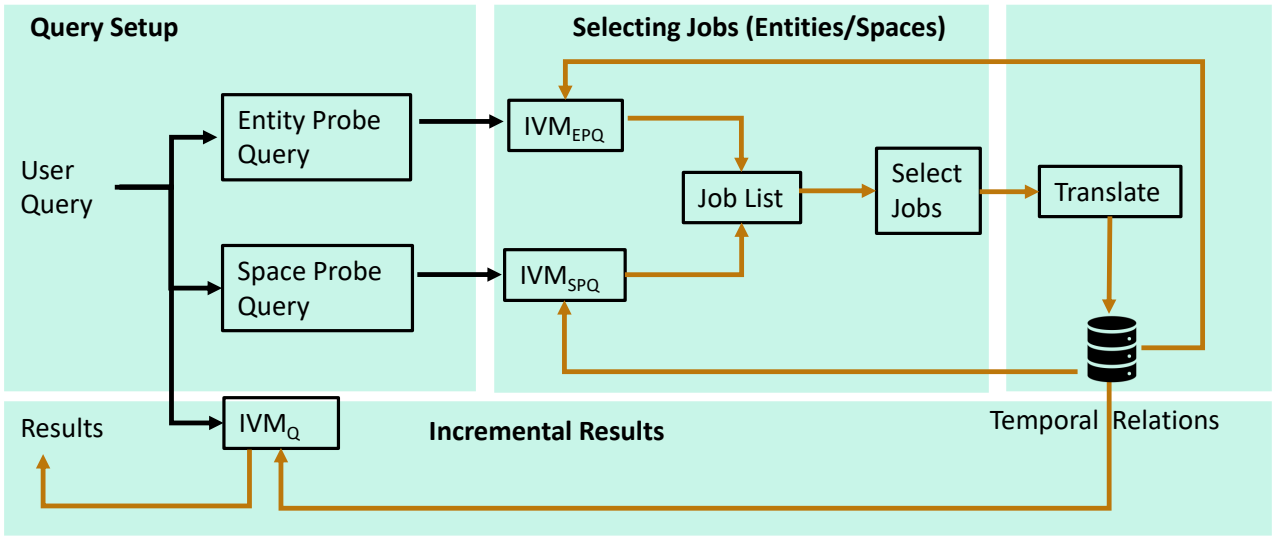


Figure 6.4: Progressive Query Processing Architecture.

is then executed to compute some of the MISSING values, resulting in updates to temporal relations.

- We return a result to the user by executing the query on the partially materialized temporal relations.
- We repeat the above steps until there is no MISSING value remaining to be computed to answer the query.

Below we describe the above approach in more detail. We will also discuss ways to implement each of the steps efficiently on existing database systems.

6.3.1 Query Setup

Before starting the epoch-based query processing, TippersDB performs a query setup/initialization phase (Lines 5- 8 of Algorithm 3). Given a query Q, we first generate the entity and space probe queries for each observable property p_j involved in the query Q i.e., EPQ_{p_j} and SPQ_{p_j}, using the technique mentioned in Section 6.2. Note that, in TippersDB's epoch-based query processing, MISSING values get computed in each epoch. As a result, the

Algorithm 3: Progressive Query Processing.

Inputs: Query Q and the duration of each epoch *epoch_duration*.
Outputs: Results of query Q after every epoch e_i

```
1  Function ExecuteQuery() begin
       QuerySetup()
2      for each epoch e_i do
3          ExecuteEpoch(e_i)
4          Results ← Run("Select * From IVM_Q")
5  Function QuerySetup() begin
6      for each p_j ∈ Q do
7          IVM_EPQ[p_j] ← GetEntityProbeQuery(Q, p_j)
8          IVM_SPQ[p_j] ← GetSpaceProbeQuery(Q, p_j)
9      IVM_Q ← Q
10 Function ExecuteEpoch(e_i) begin
11     EntityJobList ← ∅
12     SpacesList ← ∅
13     for each p_j ∈ Q do
14         EntityJobList ← Run("Select * From IVM_EPQ[p_j]")
15         SpacesList ← Run("Select * From IVM_SPQ[p_j]")
16         S_entities, S_spaces ← SelectJobsToTranslate(p_j, EntityJobList, SpacesList)
17         T_pj ← GeneratePlan(p_j, S_entities, S_spaces)
18     ExecuteTranslation(T);
```

underlying temporal relations also get updated in each epoch. With the update of temporal relations in an epoch ep_i, as a side effect, it can happen that many of the entities/spaces that were part of the output of probe queries and were not picked to be translated in the current epoch, can be discarded for translation consideration in subsequent epochs (ep_{i+1}, ep_{i+2}...) as they no longer affect the answer of the original query.

Example 6.3. *Let us assume that for the query shown in Figure 6.2a, using the probe queries, we found that the set of rooms to be covered to localize John is Room-45, Room-20, Room-100. Furthermore, the set of rooms for which the occupancy needs to be computed is Room-20, Room-100. In the first epoch, consider that we selected Room-45 to localize John and Room-20 for occupancy computation. Now in the situation where the computed occupancy of Room-20 is < 50, we can safely discard Room-20 from the set of spaces that are required to be covered to localize John.*

Therefore, it is important that we re-execute the probe queries at the beginning of each epoch

to get the latest set of spaces and entities that require translation. However, re-executing the probe queries in each epoch can add significant overhead. Therefore, instead of re-executing the probe queries, we create an *Incremental View Maintenance (IVM)* for each probe query to compute only the change in the output of the probe queries.

Background on Incremental View Maintenance (IVM). Given a view corresponding to a query q, for each table $R_i \in q$, IVM algebraically derives an incremental query Δq that is executed (e.g., using triggers as in [80]) whenever the base tables change. Δq query computes only the delta changes of the materialized view q. Correctness of IVM is characterized by ensuring that: $[q(D+\Delta D) = q(D)+\Delta q(D, \Delta D)]$, where D is an instantiation of a database, ΔD are the updates to D, $q(D)$ is the prior query results based on D, Δq is the modified query that needs to be executed on ΔD, and the notation '+' in the expression $q(D)+\Delta q(D, \Delta D)$ refers to the way of combining answers of the two queries to generate the overall answer to q over the modified data.

6.3.2 Epoch-based Execution

After setting up the probe queries, we start the epoch based query execution (Lines 10-17 of Algorithm 3). At the beginning of each epoch, first, we update the minimal set of entities and spaces to be translated using the incremental views created on the probe queries. Then we follow the following three steps.

Selecting Entities/Spaces In each epoch, we need to select a sample of entities and spaces to be translated from the minimal set of entities and spaces to be covered (computed using the probe queries). Note that, we select a subset such that the estimated cost of translation (as described in Section 5.1.2) for the selected entities is less than the epoch duration. Sample selection methods have been extensively studied for AQP [27, 100, 109]. In such systems, typically a random sample of tuples is selected based on which the approximate

aggregate values are computed. Similar to such techniques, we also choose entities or spaces or time intervals at random.

- **Sampling Entities:** We randomly select a set of entities from the output of the entity probe query for each observable property.
- **Sampling Spaces:** We randomly select a set of spaces from the output of space probe queries for each observable property.
- **Sampling time intervals:** Recall that, in Section 5.2.1 we mentioned that we divide the queried time interval into Δ intervals and generate a translation plan for each interval. In the sampling time interval strategy, we randomly select a set of Δ intervals.

Along with the sampling-based methods, we can select a set of entities and spaces using the output of the **MAYBE contextual filter functions**. Recall that, in Section 5.3.2 we described MAYBE contextual filter functions that take as input a set of entities, spaces, and a time interval and return a list of entities and spaces that should be prioritized for translation before other entities and spaces.

Translation After selecting the set of entities and spaces to translate, we generate a translation plan for the selected entities and spaces. We then execute the translation plan to compute the MISSING values and update the underlying temporal relations.

Incremental Computation We can simply execute the query at the end of each epoch to compute the new answer set. However, executing the complete query in each epoch will result in high overhead and therefore is not feasible. Instead, we should execute the query progressively such that only the delta answers based on temporal relations modified due to translations are computed. For this purpose, we create an incremental materialized view denoted as IVM_Q for the original query Q. Users can fetch complete query results at the end of an epoch by querying the IVM_Q. If the complete answer set is large, users can

Figure 6.5: Progressive results at different levels of hierarchy.

retrieve delta changes of answers, *i.e.*, inserted/deleted/updated tuples from the previous epoch. The current implementation allows users to fetch delta answers only from the last epoch. Fetching delta answers from any arbitrary epoch using a cursor is complex (will be supported in a future version), since the query processing in both designs is not demand-driven, as in SQL databases.

6.4 Exploiting Hierarchy for Progressiveness

In Section 5.3.1, we showed that in many cases, for observable properties having a hierarchical data types, it is possible that the translation at a higher/coarser level level of granularity is less expensive compared to translation at a lower/finer level, and therefore, hierarchical data types can be used to reduce the amount of translation to be done. In this section, we develop a method to use hierarchical data types not only to reduce the number of translations but also to provide users with early answers at coarse levels. For instance, consider a query asking for rooms that John visited on a given day. As we have shown in Section 5.3.1, localization at the floor/region level is much cheaper than localization at the room level. Therefore, instead of directly returning the set of rooms that John has visited, we can first return the regions that John has visited, and then progressively return the set of rooms that John has visited (as shown in Figure 6.5).

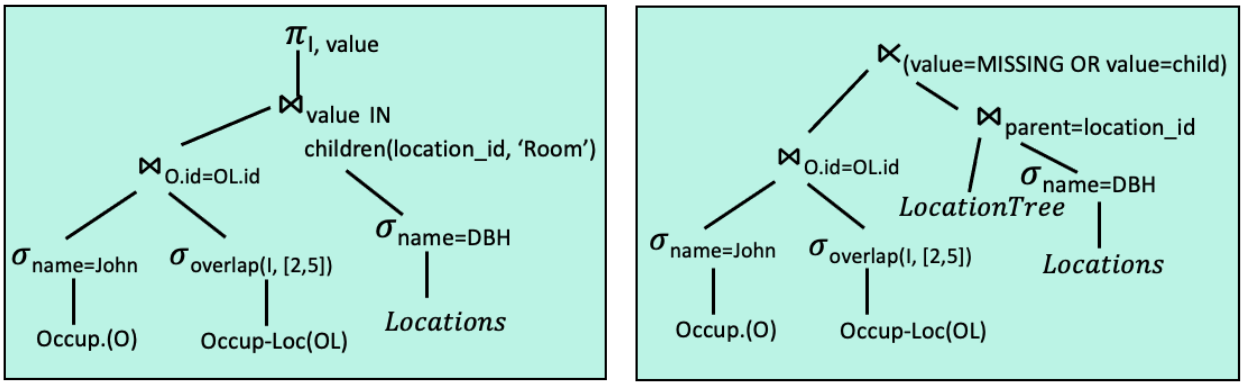
Location__id	name	Parent	Space_type_id
1	DBH	NULL	1
2	Floor1	1	2
3	Room1	2	3
4	Room2	2	3

(a) Locations

parent	child
1	1
1	2
1	3
1	4
2	2
2	3
2	4
3	3
4	4

(b) LocationTree

Figure 6.6: Locations and LocationTree tables



(a) Original Query (b) Entity probe query for OccupantLocation relation.

Figure 6.7: Probe query generation for hierarchical data types.

6.4.1 Probe Queries

Recall that in Section 6.2.1, we described that for translation we only need to consider those entities that have MISSING values. However, for an observable property with the hierarchical data type, it is possible that an entity has a value at a coarser level stored in the temporal relation, computed as a result of the translation done during previous queries or epochs. Therefore, along with the entities with MISSING values, we need to consider those entities as well for which the value of the observable property is at a coarser level.

Example 6.4. *Consider the query shown in Figure 6.7a that finds all the rooms that 'John' visited in the DBH building during the time interval [2, 5]. In this query along with those*

tuples of 'John' in OccupantLocation relation that have MISSING values, we have to consider those tuples also where John's location is already computed at a coarser level than the room level i.e., region, floor or building as long as the location is inside DBH.

In general, to generate an entity probe query for an observable property p_j of hierarchical data type, along with the steps mentioned in section 6.2.1, we perform the following two more steps.

- *Expanded Tree Table*: Using a recursive CTE, we create a table that stores for each node in the hierarchy of the data type of property p_j all its children. We denote such a table by $Tree_{p_j}$. For example, for the *Locations* table containing all the spatial regions, a *LocationTree* table is created containing for each spatial region all its children.

- *Rewriting selection and join conditions*: We rewrite the selection condition of type $R_i p_j.value = a_1$ as $(R_i p_j.value = MISSING \vee R_i p_j.value = Tree_{p_j}.child) \wedge Tree_{p_j}.parent = a_1$. Similarly, a join condition of type $R_i p_j.value = R_k.A_1$ is rewritten as $(R_i p_j.value = MISSING \vee R_i p_j.value = Tree_{p_j}.child) \wedge Tree_{p_j}.parent = R_k.A_1$. Figure 6.7b shows the entity probe query generated for the query in Figure 6.7a. In the query we can see that the condition OL.value=location_id is rewritten as (OL.value=MISSING OR OL.value=LocationTree.child) AND LocationTree.parent=location_id.

6.4.2 Epoch-based Execution

Selecting entities/spaces: For hierarchical observable properties, we modify the sampling strategies mentioned in Section 6.3.2. Instead of selecting entities randomly, we first prioritize entities having MISSING values and translate them at a coarser level. Once no entity have MISSING values, we sample from entities having values at a coarse level and translate them

at a finer level. We keep on sampling entities based on the granularity of their value using the above steps until all the entities have values at the granularity required by the query. currently selecting select entities having values at a coarser level before selecting the entities having values at a finer level.

Example 6.5. *Consider a query asking for the current location of all graduate students. For this query, we will first sample from students for whom the location is completely MISSING and and for them we will compute the building level location. If there is no student left for whom the location is MISSING, we will sample from the students who have location at the building level and for them compute the location at the region level. Finally, when there is no student left for whom the location is at the building level, we will sample from the students who have location at the region level and for them we will compute the location at the room level.*

Translation After selecting the set of entities and spaces to translate, we generate a translation plan for the selected entities and spaces. Note that, for the hierarchical data types we pass the level of granularity to the translation operator at which the translation is required to be done.

Incremental Computation: To allow answers at a coarser level to be returned, we cannot directly run the original query on the partially materialized temporal relations. We need to make sure that we re-write the selection and join conditions on the hierarchical observable properties such that along with the tuples that directly satisfy the condition, the tuples that will satisfy the same condition when written at a coarse level are also passed. In particular, to generate the result query we rewrite the selection condition $R_i p_j.value = a_1$ as $(R_i p_j.value = parent \vee R_i p_j.value = Tree_{p_j}.child) \wedge (Tree_{p_j}.parent = a_1 \vee Tree_{p_j}.child = a_1)$. For example, Figure 6.8 shows the IVM query generated for the query shown in Figure 6.7a.

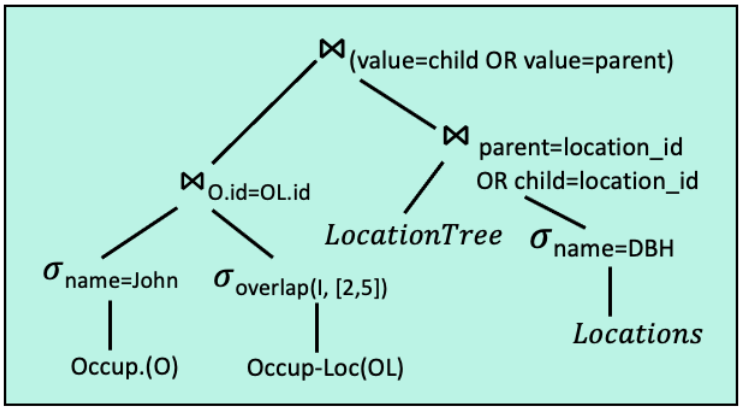


Figure 6.8: IVM query for hierarchical data types.

6.4.3 Quality

In the previous section we showed that, TippersDB can return fast answers at coarser level first before returning answers at a finer level. Consider a query asking for room level location of 'John'. In such a query, the system may return coarser level results e.g., region or building level location of John. In such a situation we cannot use the F_1 measure to compute the quality of answers returned at the end of an epoch. We need a quality metric that can give credit for partially correct answer i.e., an ancestor of the actual value. Therefore, we define a modified quality metric based on the hierarchical precision, recall and F_1 measure as follows.

$$
\begin{aligned}
HF_1(Ans_w) &= \frac{HPre(Ans_w) \cdot HRec(Ans_w)}{(HPre(Ans_w) + HRec(Ans_w))} \\
HPre(Ans_w) &= \frac{\sum_{i=1}^{Ans_w} |Children(Ans_w[i]) \cap Children(Ans^{real}[i])|}{\sum_{i=1}^{Ans_w} |Children(Ans_w)|} \\
HRec(Ans_w) &= \frac{\sum_{i=1}^{Ans_w} |Children(Ans_w[i]) \cap Children(Ans^{real}[i])|}{\sum_{i=1}^{Ans_w} |Children(Ans^{real})|}
\end{aligned}
\tag{6.4}
$$

where $Children(x)$ represents the set of all the child nodes of the node represented by x.

Sensor	No.	Rows (M)	Size (GB)	Observing Functions (Cost)
WiFi	300	80	76	location(room:150ms,region:10ms) occupancy(room:100ms, region:8ms)
WeMo	1,400	37	4	energy(room:20ms)
Hvac	250	15	10	energy(region:50ms)
Camera	1,400	20	840	location(room:200ms); occupancy(room:120ms)
GPS	5,000	50	10	building-location (5ms)
Watch	5,000	50	20	vitals (18ms)

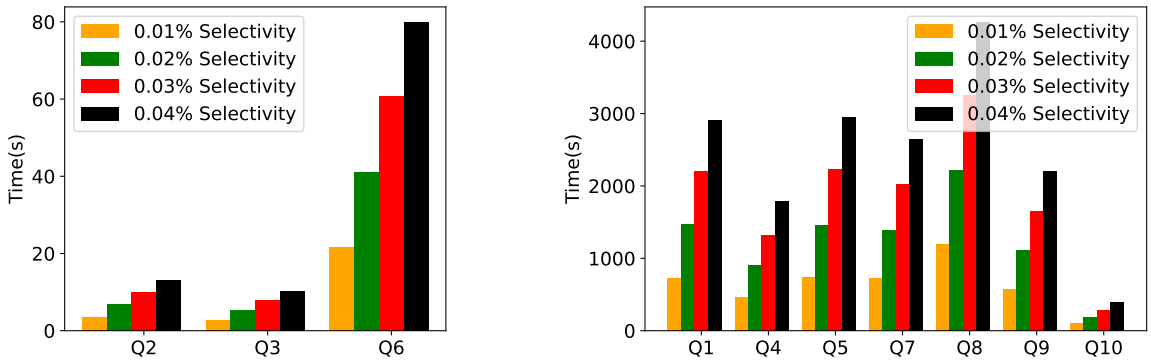
Table 6.1: Sensor Dataset and Functions.

Q1	Fetch room-location of John during time (t_1, t_2)
Q2	Find all people in room r during time (t_1, t_2)
Q3	Fetch occupancy of room r during time (t_1, t_2)
Q4	Fetch all rooms with occupancy greater than 100 during time (t_1, t_2)
Q5	Fetch all users co-located (same room, same time) with John during (t_1, t_2)
Q6	Retrieve users who went from room r_1 to r_2 during (t_1, t_2)
Q7	Retrieve average time spent per room by users in different types of rooms during (t_1, t_2)
Q8	Find occupancy of all rooms visited by John during (t_1, t_2) and with occupancy > 50
Q9	Find rooms consuming > 100 energy units with average occupancy < 50 during (t_1, t_2)
Q10	Find all users who were healthy and visited buildings that were visited by unhealthy individuals prior to them between (t_1, t_2)

Table 6.2: Queries.

6.5 Experiments

For the experiments, we use the smart campus scenario. We consider that the buildings inside the smart campus are instrumented with WiFi APs, cameras, power meters (WeMo devices [17]), and HVAC sensors. Also, we assume that the campus is inhabited by various people and the majority of people carry a GPS-enabled smartphone and a smartwatch. As mentioned in §4.2, the OER model in this scenario consists of people and rooms as main entities. In the model, people have location and vitals as observable properties, and rooms have occupancy and energy consumption as their observable properties. The location property is hierarchical (as discussed in §4.1). The granularity of time for the sensor data and time intervals in temporal relation is set to one second.



(a) Q2,3,6 (b) Q1,4,5,7-10

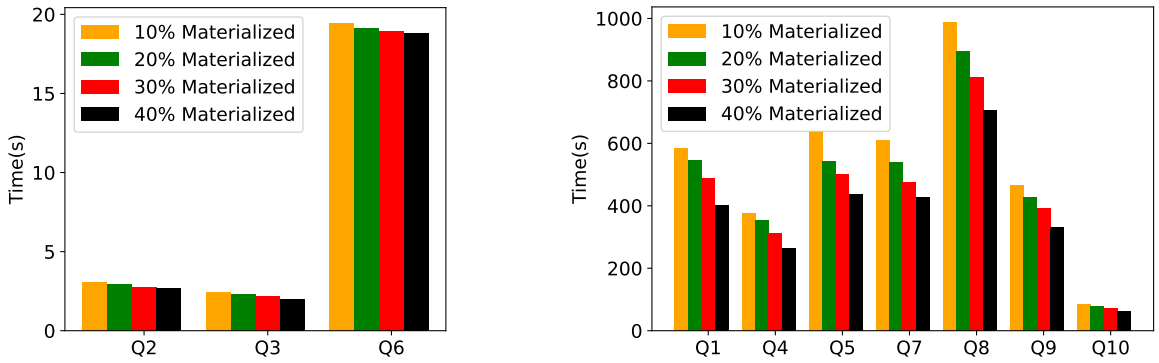
Figure 6.9: Effect of selectivity on query execution time.

Dataset. We used the sensor data generation tool provided in [68] to generate synthetic sensor data for a month for a campus with 25 buildings. Table 5.3 shows the number of instances of each sensor type, the number of rows, and the size of the generated sensor data.

Queries. We selected the following ten queries; see Table 5.4: Queries Q5-Q8 are extracted from SmartBench [68], an IoT database benchmark. All queries are expressed on the semantic model referring to entities and their observable (or not) attributes.

Observing Functions: We use observing functions to compute building location/occupancy from GPS data, region and room location/occupancy from WiFi data, and room location/occupancy from camera data. Similarly, the energy usage of a room is computed using WeMo data and at region level using HVAC data. We use smartwatch data (i.e., heart rate, O_2 level) to generate a health report of a person. The cost of each function is given in Table 5.3.

The following experiments were performed on a server with 16 core 2.50GHz Intel i7 CPU, 64GB RAM, and 1TB SSD. Both the sensor data and the temporal relations were stored in PostgreSQL.



(a) Q2,3,6 (b) Q1,4,5,7-10

Figure 6.10: Effect of materialization on query execution.

6.5.1 TippersDB Performance

In the following experiments we study the effect of selectivity and materialization on the latency of a query executed using TippersDB, but without the epoch-based execution i.e., all the entities and spaces that are in the output of the probe queries are translated together.

Effect of selectivity: This experiment studies the effect of selectivity on the query execution time of different queries. Figure 6.9 shows the execution time of different queries under different level of selectivity. We vary the selectivity of queries by varying the length of time interval (t_1, t_2) parameter in the queries. Observe that, as expected for all queries, the query execution time increases with an increase in query selectivity as the system has to do more translations.

Effect of Materialization: This experiment studies the effect of materialization level (the percentage of data that has non MISSING values) of temporal relation on query execution. Figure 6.10 shows the execution time of different queries under different levels of materialization. For all queries, we can see that the higher the materialization level lower the query execution time, which is expected since probe queries will filter all those entities and spaces that already have the value of the observable property computed.

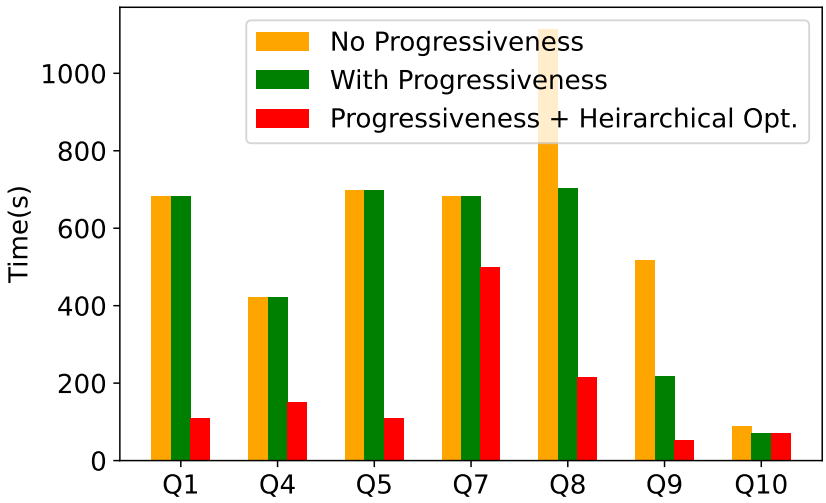
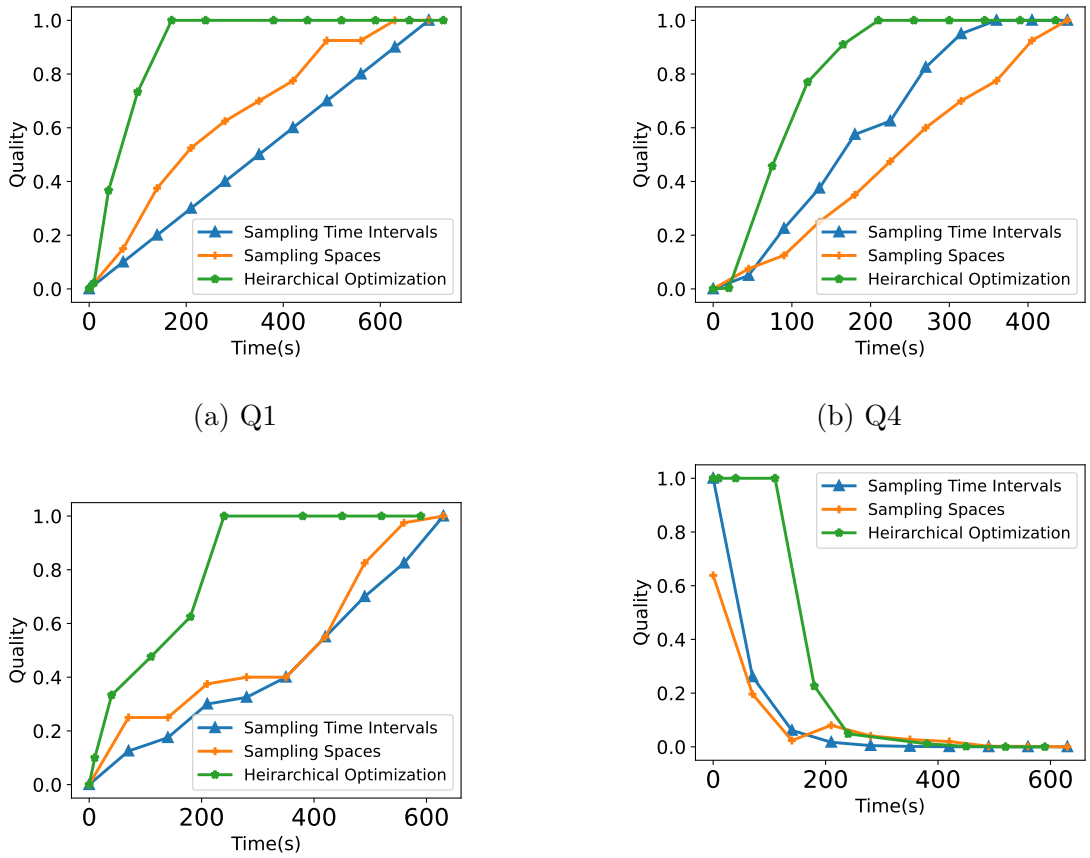


Figure 6.11: Total query execution time with different strategies

6.5.2 Progressiveness

The following experiments evaluate the the effectiveness of progressive query processing approach in reducing the wait time for the users.

Query Latency with and without Progressiveness: Figure 6.11 shows the total execution time of queries in TippersDB under three different settings: a) no progressiveness i.e., a translation plan is generated for all the entities, and spaces that are in the output of probe queries; b) epoch-based progressive query execution; c) epoch-based progressive query execution including optimization for hierarchical observable properties. Observe that for queries Q8, Q9, Q10 that involve more that one observable property and joins between them, the total execution time with the progressive approach is less than the total execution time with no progressiveness setting. This reduction in query execution time is due to the removal of entities and spatial regions from the plan space through the incremental views created on the probe queries. Furthermore, almost all queries benefited extensively from hierarchical optimization. For instance, the total query time for Q1 and Q4 reduced from 670s to 90s and from 410s to 184s, respectively.



(a) Q1

(b) Q4

(c) Q5

(d) Q7

Figure 6.12: Progressive improvement of quality of different queries.

Number of Translations: Table 6.3 shows the total number of translations done by TippersDB for each query with/without progressiveness and with/without hierarchical optimization. We compute the number of translations as the sum of the number of entities and the number of spaces over all delta time intervals for which a translation plan was generated. Observe that for many queries the progressive approach reduced the number of translations. Furthermore, the number of translations reduced significantly with hierarchical optimization. For instance, for query Q1 and Q4 the number of translations reduced from 198K to 32K and from 145K to 48K respectively.

Progressiveness Achieved: In this experiment we evaluate the different progressive approaches in terms of progressive quality improvement achieved. Figure 6.12 shows the quality

Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
No Prog.	198K	30	30	145K	198K	60	198K	343K	290K	396K
With Prog.	198K	30	30	145K	198K	60	198K	230K	116K	340K
Prog. with Optimizations	32K	30	30	48K	32K	60	144K	74K	38K	340K

Table 6.3: Number of translations.

Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Translation	93%	81%	80%	93%	91%	81%	95%	92%	89%	91%
Planning	6%	15%	15%	6%	8%	14%	4%	6%	9%	8%
DB Queries	1%	4%	5%	1%	1%	5%	1%	2%	2%	1%

Table 6.4: TippersDB overhead.

with respect to time for queries Q1, Q4, Q5, and Q7 using three different progressive approaches a) sampling spaces, b) sampling time interval c) hierarchical data type optimization. Note that the quality of answers for set based answers are computed using F_1 measure and hierarchical F_1 measure (for results with hierarchical data types) and using root mean square error for aggregation queries. Figure 6.12 shows that all the approaches are able to provide users with high quality results early during the query execution and therefore reduced the wait time for the user. Furthermore, the progressive approach with hierarchical data types performed the best and was able to provide users with answers that improved faster as compared to the progressive approaches without the hierarchical optimization.

System Overhead: Table 5.6 shows for each query Q1-Q10, the percentage of the total time of query execution spent in translation, generating translation plans, and executing incremental view-based probe and result queries on the underlying database. For most queries, more than 90% of time is spent in translation and only a small part of the total time is spent in translation planning and running queries on the database.

Chapter 7

Case Study

In this chapter we show TippersDB's usability through different IoT deployments. We show how applications from different domains can be modeled using the TippersDB Observable ER model.

7.1 Assisted Living Space

A large number of older adults live in assisted living spaces. An IoT system for assisted living spaces provides care givers access to the information about an individual's changing health conditions, their personalized needs and identifies those in need of specialized triage and critical care. It provides access to information about the living facilities (e.g., floor plans, operational status, number of residents) and about the residents, for example, health conditions such as need for dialysis, oxygen, and personal objects to reduce anxiety. It can also empower first responders to improve response outcomes during disasters.

The assisted living smart space setting, which is a part of the CareDEX project [21], aims to ensure the safety of older adults who require personalized care. Here, senior housing facilities

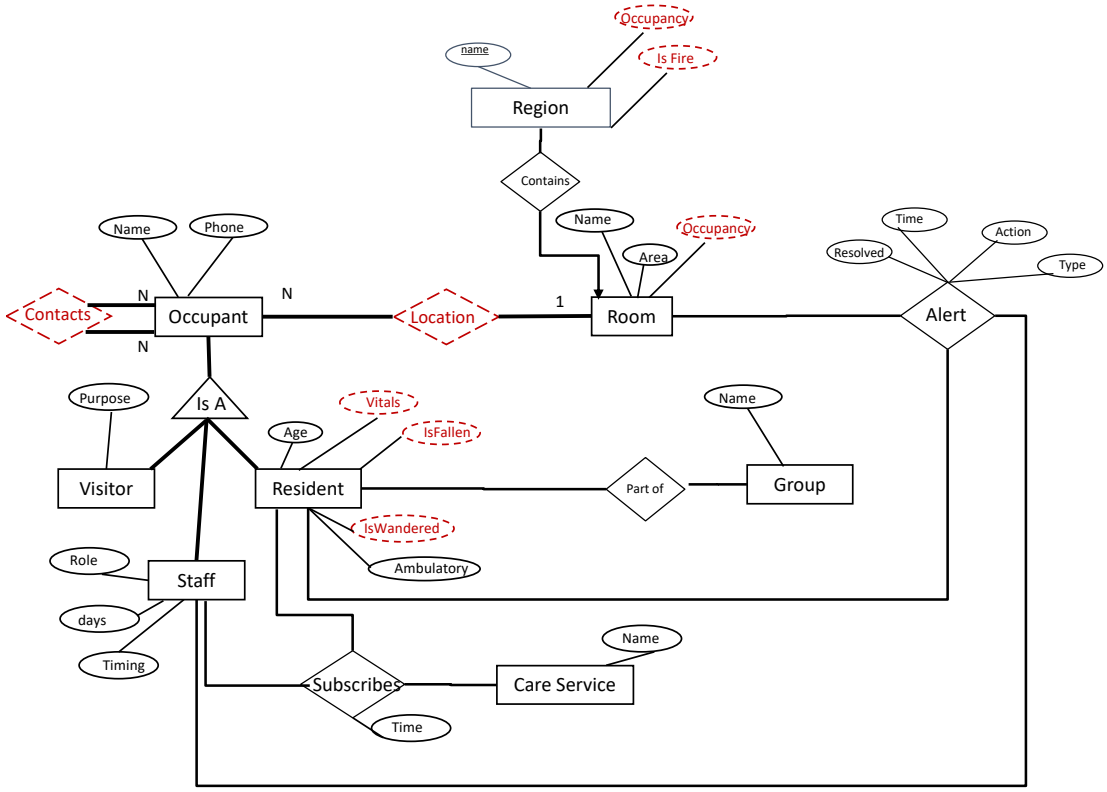


Figure 7.1: OER model for an assisted living space showing entities with observable attributes/relationships (in red).

and residents are instrumented with diverse sensors (fall detection, wander alerts, motion detectors) enabling personalized monitoring of residents by caregivers to identify those in need of urgent care. Information generated from in-situ motion sensors and WiFi access points are merged with mobile fall detection or wander alert sensors in TippersDB to create improved awareness applications for caregivers. TippersDB applications will support analysis to pinpoint unsafe regions where most falls have taken place, identify isolated residents who exhibit low levels of interaction with others, and track residents who are an elopement risk and at danger of leaving the facility. An OER model for an assisted living scenario is given in Figure 7.1. Along with showing different entities and relationships between them, it shows the observable attributes and relationships, for example resident entity set has the *isFallen* attribute marked as observable. We have created some prototype dashboards for the deployments. Figure 7.2 shows as spatial region of an assisted living space modeled using TippersDB, Figure 7.3 shows coverage of different WiFi access points in the space

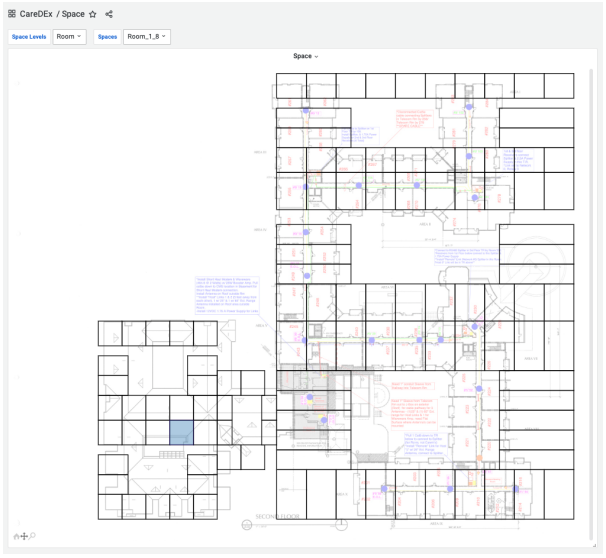


Figure 7.2: Screenshot of space model of a assisted living space.

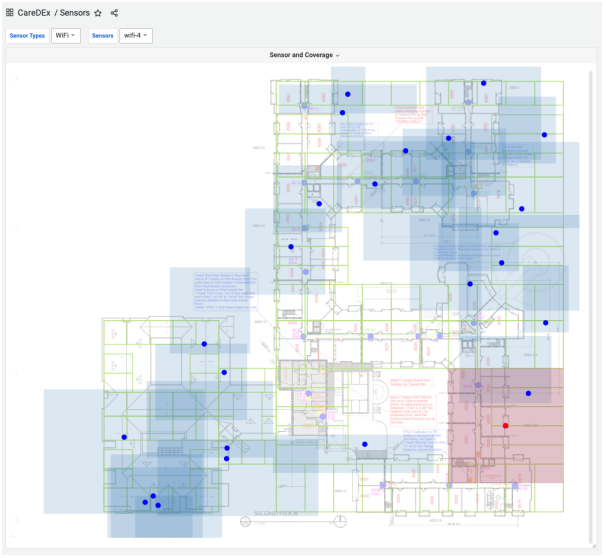


Figure 7.3: Screenshot of coverage of WiFi access points in an assisted living space.

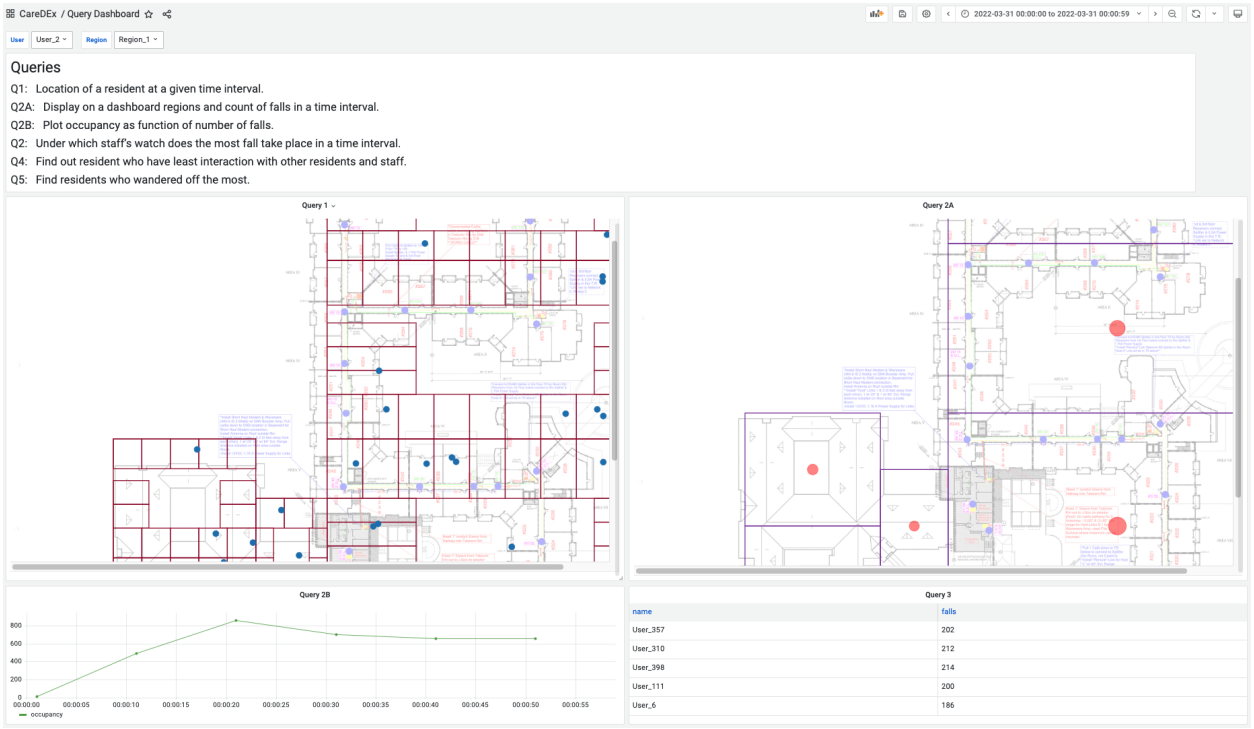


Figure 7.4: Screenshot showing a dashboard running various queries in the assisted living smart space scenario.

and Figure 7.4 shows a dashboard running several queries on the OER model.

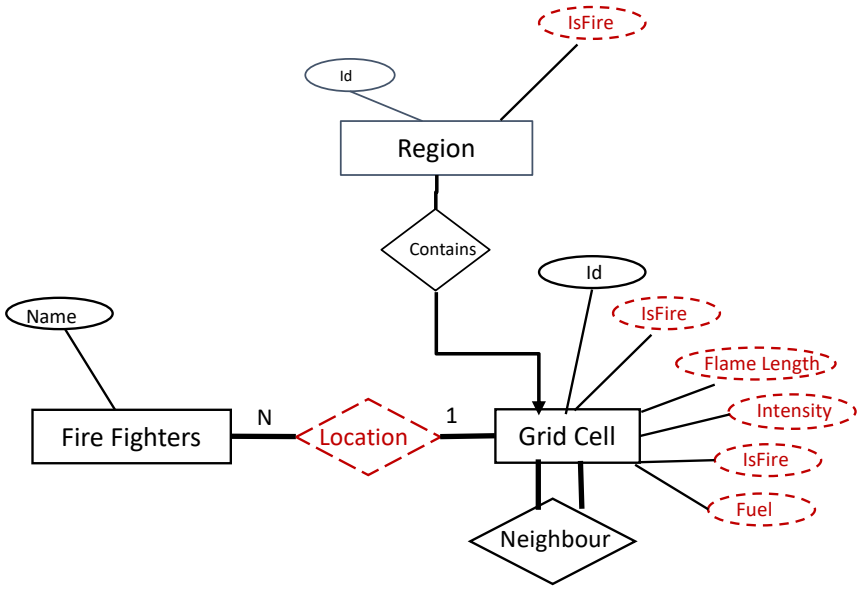


Figure 7.5: OER model for a prescribed fire exercise showing entities with observable attributes/relationships (in red).

7.2 Prescribed Fire Monitoring

We also used TippersDB in the context of *Prescribed Fire Monitoring* as part of the SPARx-Cal project [26]. Prescribed Fires are planned, intentional and controlled applications of fire to a land area, under specified predicted weather conditions - it is used as a proactive technique to prevent rapid spread of wildfires in forests and wild-land urban communities. Prescribed fires run the risk of escaping; fast and accurate monitoring of their progress is critical. In SPARx, a range of devices/sensors are used to monitor burn progress, drones are used to capture aerial imagery of fire levels while insitu sensors at the burn site capture environmental parameters (wind, smoke, air quality). Thermal and RGB images from drones and insitu cameras are analyzed for fire presence, flame length and fire intensity in different regions. Simultaneously, wind sensors (wind direction and speed), air quality sensors (levels of particulate matter and smoke) and humidity sensors provide local environmental conditions that dictate fire progress. GPS devices carried by personnel are used for crew location to direct them to regions where attention is required for ignition or control. Through a TippersDB deployment, an analyst will be able to pose queries to better monitor

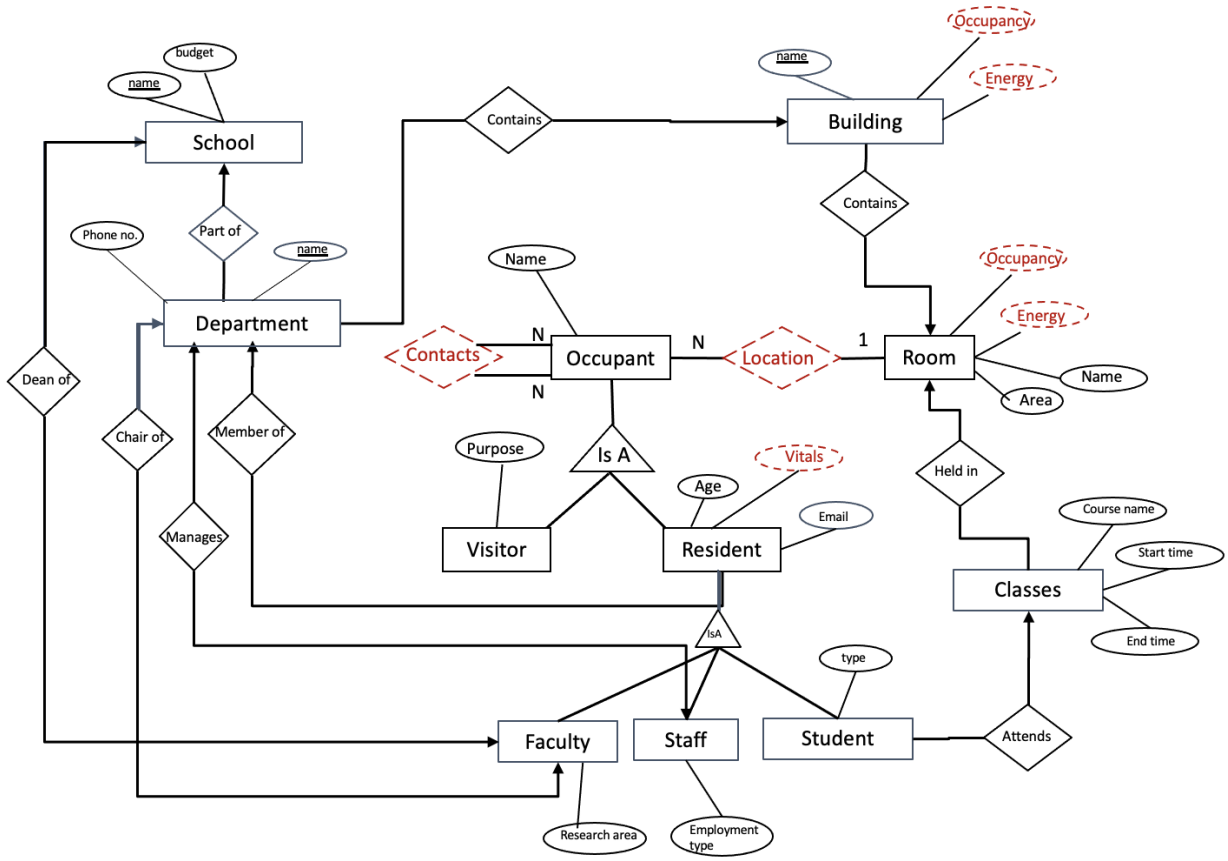


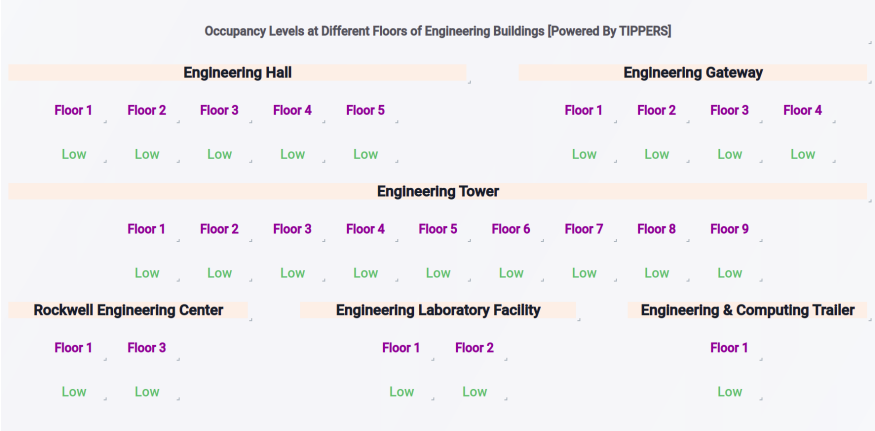
Figure 7.6: OER model for a smart campus showing entities with observable attributes/relationships (in red).

the burn (identify regions with fire presence/absence, determine regions with unfavorable wind/humidity conditions, locations of fuel with low humidity) and control further data collection through drone path planning. An OER model for a prescribed fire scenario is given in Figure 7.5.

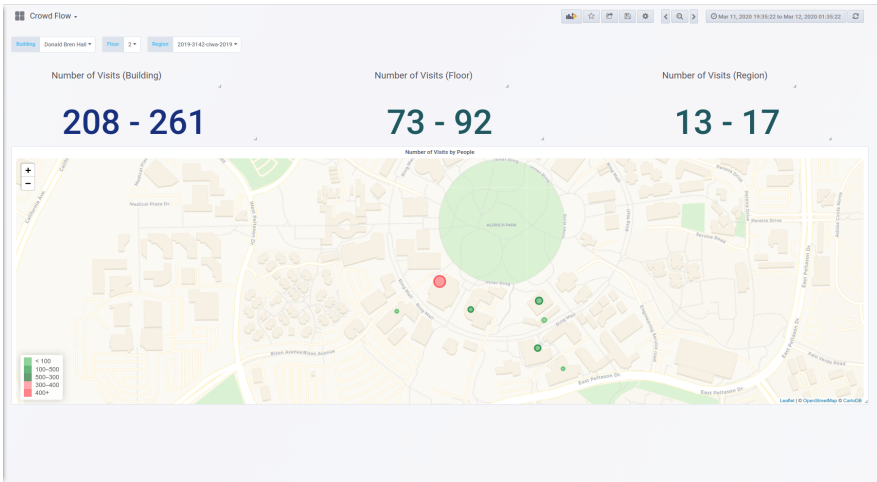
7.3 Smart Campus

Another use case of TippersDB is a *Smart Campus* deployment. A smart campus is instrumented with variety of sensors to help improve ways in which people interact with the campus infrastructure and among themselves. TippersDB deployment at a smart campus

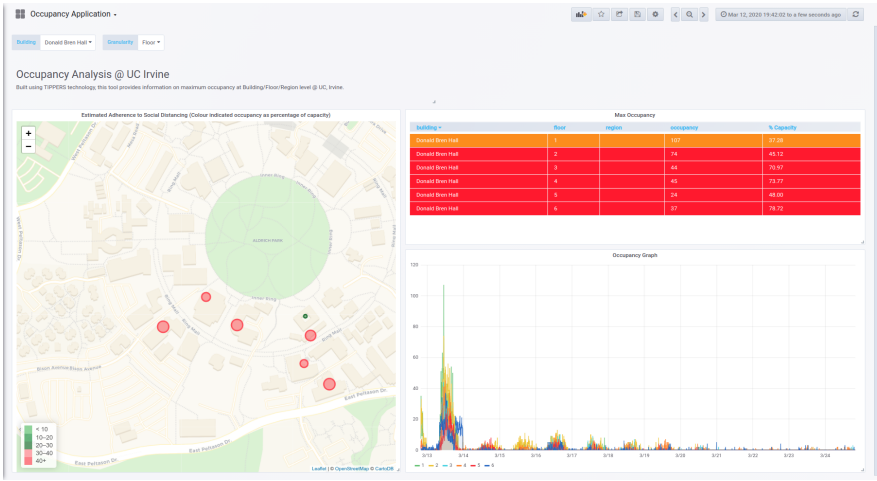
provide several services/applications to the users as well as campus administrators. These services could correspond to locating group members in real-time, customizing heating, ventilation, and air conditioning (HVAC) controls based on user preferences, contact tracing by computing who came in contact with whom and when, monitoring occupancy of different parts of a building over time, analyzing building usage over time, understanding social interactions amongst residents and visitors, and keeping track of facilities visited by visitors. To enable these services, a smart campus supports diverse type of sensors ranging from HVAC sensors (e.g., temperature, humidity, and pressure sensors), motion and activity sensors, Bluetooth sensors, cameras, WiFi access points. Note that in most of the smart campus applications listed above, localization of users plays an important role, therefore location sensing technologies are an integral part of such a deployment. Location sensing technologies can broadly be classified as: *active* or *passive* based on whether it requires a user to carry new hardware, download software, and/or participate in the localization process, or if the infrastructure can determine the user's location without his/her active participation. Active technologies include GPS on mobile devices carried by a user, WiFi or cellular signal strength triangulation, and inertial sensing. Passive techniques include camera-based localization as well as localization-based on connectivity events in the WiFi network. Also, the campus supports GPS-based localization of individuals who have downloaded an application to transmit their GPS coordinates from their mobile phones to the system. The OER model shown in Figure 7.6 represents different entities, relationships along with corresponding observable properties in a smart campus scenario. Figure 7.7 shows screenshots of different applications built using TippersDB for a smart campus.



(a) Engineering buildings dashboard.



(b) Crowd-Flow application.

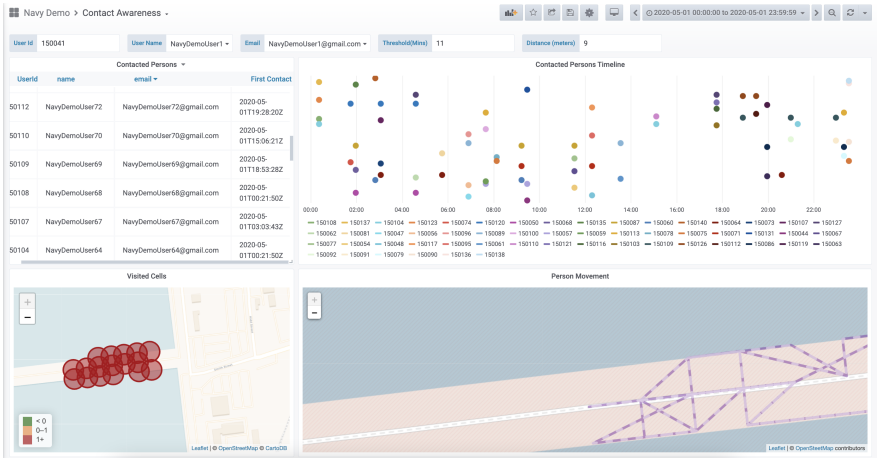


(c) Occupancy application.

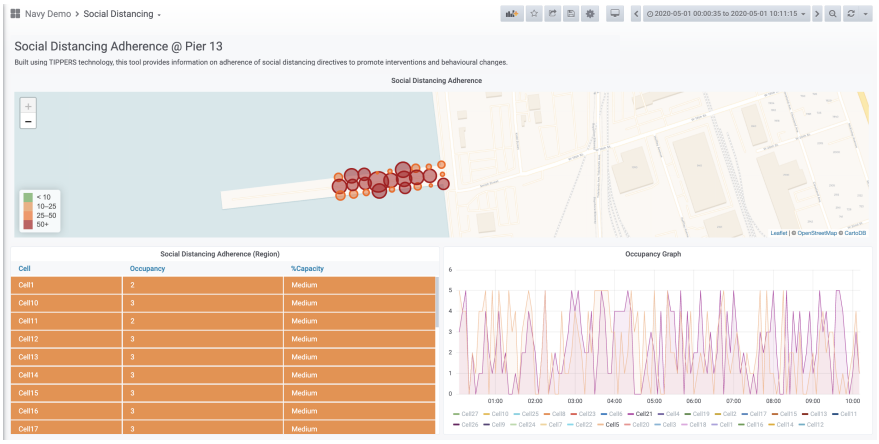
Figure 7.7: Smart space applications built using TippersDB.

7.4 Naval Base

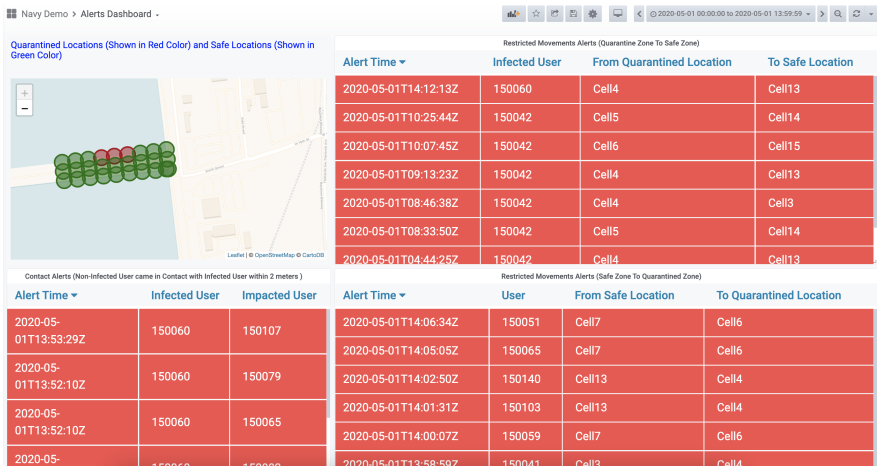
Another use case of TippersDB is a deployment at a pier in Naval Base San Diego [41], the goal was to perform fine-grained tracking of people. The space, a pier at the naval base, was instrumented with WiFi APs, Bluetooth beacons, RFID readers, and smartphones with a client application (capturing GPS readings among others). The challenge in this deployment was to accurately locate individuals so that requirements related to social distancing (in terms of minimum distance between two persons) can be fulfilled. This required data form all the above mentioned sensors to be fused together to localize people accurately and robustly. The experiments performed (tailored towards COVID-19 mitigation) involved 30 participants, who performed a series of scenarios as they moved between different zones on the pier. Data was passively captured using the WiFi infrastructure and and actively using the client application on the smartphones. Then, the data was correlated by to locate individuals in the different zones of the pier (each zone had been defined in the system to cover a small area). In this process, observing functions using machine learning techniques were used to triangulate the location of a person given the signal obtained from the Bluetooth beacons, WiFi APs and GPS data. Finally, the information of multiple individuals had to be correlated to check whether social distancing criteria were met. Several applications were build using TippersDB. One example is the quarantine zone application, that monitors which areas have been visited by infected individuals and raise alerts when an infected individual visited a safe location. Another example is the "secure bubble" transfer application, in which a group of individuals have to be protected and alerts have to be triggered if any of its members came in contact with an infected individual. Figure 7.8 shows screenshots of different applications built using TippersDB for the naval base scenario.



(a) Contact Awareness.



(b) Social Distancing.



(c) Alerts Dashboard.

Figure 7.8: Naval base applications built using TippersDB.

Chapter 8

Conclusion And Future Work

In this chapter we present the conclusions of the work presented in this thesis on developing a middleware based system, designed to build smart-space analytical applications. We also discuss some of the future directions of research related to the work presented in this thesis.

8.1 Conclusions

In this thesis, first we introduced the SmartBench benchmark which is derived from a deployed smart building monitoring system. We described SmartBench's extensible schema that captures the fundamentals of an IoT smart space and. We listed a set of eleven benchmark queries and described the motivation behind each of them. We described, the data generation tool of SmartBench that generates large amounts of synthetic sensor and semantic data based on seed data collected from a real system. We presented an evaluation of seven database systems (representing different database technologies) using the SmartBench benchmark. We highlighted key findings that can be considered when deciding what database technologies to use under different types of IoT workloads. Using the results of the

benchmark we made a case for the need of data virtualization query-driven translation.

Then, we introduced the TippersDB data model that provides data virtualization through semantic abstraction. We showed how the data model hides the complexities of sensor infrastructure and sensor data processing codes from the application developers and provides them with an interface to write applications on top of higher-level semantic data. We described the realization of TippersDB data model on top of an existing database system. We described the translation mechanism of TippersDB and showed how TippersDB integrates query processing with translation. We discussed how TippersDB reduces the number of translations by removing redundant translations using the query context. We also introduced an optimization that exploits the hierarchical nature of certain data types to further reduce the number of translations.

Finally, we presented progressive query processing techniques to further reduce query latency and to provide early results to the users. We described the semantics of progressive query processing and ways to provide progressive answers. We described how TippersDB selects data to be translated, exploits hierarchical data types (e.g., location), and computes incremental answers.

8.2 Future Work

Self Driving Translation: In this thesis we explored query-driven sensor data translation. Query-driven translation makes data available for analysis early and removes redundant translation. However, depending upon the resource availability, it is possible that some data can be translated at the ingestion time itself. In this case, the system, based on the resources and workload, should carefully select the subset of the sensor data (along with the observing functions) that should be translated at the ingestion time, such that the later queries can

benefit the most. Therefore, a possible future work is to develop a self-driving engine that figures out what data should be translated at the ingestion, in the background (as an offline process) or at the query time.

Native Implementation of TippersDB: Our focus in this thesis was on designing mechanisms for sensor data translation and progressive query processing techniques layered on top of existing database systems, treating them as essentially black boxes. While such an approach offers flexibility and an easier path to the adoption of technology, mechanisms to modify the underlying storage, indexing, and query processing mechanisms to support sensor data translation natively in database systems with the goal of exploring how much improvement in performance can result by modifying the underlying technology is worthy of exploration in the future.

Privacy: In this thesis we explored techniques to map/translate an application level query to sensor data. However, it is important that the system only uses the sensor data that is allowed for the query getting executed. Furthermore, the system should allow the applications to create policies at the higher semantic level and implicitly translate them on to the sensor data. Therefore, enhancing TippersDB to support semantic as well as sensor level policies, making it GDPR [130] compliant, designing an efficient policy aware translation mechanism are all interesting future work.

Supporting what-if analysis: In this thesis we focused on supporting real-time as well as analytical IoT applications. Exploratory analysis e.g., what-if analysis is also an important part of data analytics. A possible future work is to explore mechanisms to perform progressive what-if analysis inside database systems.

Multi-query optimization: In this thesis, we have considered optimizing sensor data translation in the context of a single query, executed at the TippersDB server. However, it is possible that multiple queries requiring overlapping translations might arrive at the same

time in the system. In this case, the translations done for one query affect other queries and therefore, a technique that generates a translation plan keeping other queries in consideration is needed.

Edge computing: In this thesis, we considered a model, where sensor data is collected from sensors and then processed (at query-time) at a centralized server. Many sensor devices themselves have capability to store and process data to a certain limit. Enabling the compute and storage capabilities of such devices with TippersDB by supporting mechanisms to run observing functions directly on the sensing device is an interesting direction of future work.

Bibliography

[1] Amazon redshift data warehouse. https://aws.amazon.com/pm/redshift/.

[2] Apache kafka. `https://kafka.apache.org/23/documentation/streams/`.

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

[4] Aws iot. `https://aws.amazon.com/iot/`.

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

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

[7] Griddb. https://griddb.net/en/.

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

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

[10] Nest. `http://www.iot-nest.com/`.

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

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

[13] Snowflake data warehouse. https://www.snowflake.com/workloads/data-warehouse-modernization/.

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

[15] Tippersdb extended version. `https://github.com/ucisharadlab/tdb`.

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

[17] Wemo. `https://www.wemo.com`.

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

[19] Influxdb system. `https://www.influxdata.com/time-series-database/`, 2018.

[20] Timescale system. `https://www.timescale.com/`, 2019.

[21] Enabling disaster resilience in aging communities via a secure data exchange. https://sites.uci.edu/caredex/, [Online; accessed May-2022].

[22] Oracle erp. https://www.oracle.com/erp/, [Online; accessed May-2022].

[23] Oracle financials. https://www.oracle.com/erp/financials-cloud/, [Online; accessed May-2022].

[24] Salesforce crm. https://www.salesforce.com/crm/, [Online; accessed May-2022].

[25] Sap s4/hana erp. https://www.sap.com/products/s4hana-erp.html, [Online; accessed May-2022].

[26] Sparx cal: Smart practices and architectures for rx fire in california. https://sites.uci.edu/sparxcal/, [Online; accessed May-2022].

[27] S. Agarwal et al. Blinkdb: Queries with bounded errors and bounded response times on very large data. EuroSys '13.

[28] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

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

[30] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. *Proc. VLDB Endow.*, 7(11):999–1010, 2014.

[31] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progresser: Adaptive progressive approach to relational entity resolution. *ACM Trans. Knowl. Discov. Data*, 12(3):33:1–33:45, 2018.

[32] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. Query: A framework for integrating entity resolution with query processing. *PVLDB*, 9(3), 2015.

[33] M. P. Andersen and D. E. Culler. Btrdb: Optimizing storage system design for time-series processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

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

[35] D. Archer, M. A. August, G. Bouloukakis, C. Davison, M. H. Diallo, D. Ghosh, C. T. Graves, M. Hay, X. He, P. Laud, et al. Transitioning from testbeds to ships: an experience study in deploying the tippers internet of things platform to the us navy. *The Journal of Defense Modeling and Simulation*, 19(3):501–517, 2022.

[36] G. Ariav. A temporally oriented data model. *ACM Transactions on Database Systems (TODS)*, 11(4):499–527, 1986.

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

[38] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *2015 ACM SIGMOD Int. Conf. on Management of Data*, pages 1383–1394. ACM, 2015.

[39] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 583–598, 2016.

[40] M. August, C. Davison, M. Diallo, D. Ghosh, P. Gupta, C. Graves, S. Han, M. Holstrom, P. Khargonekar, M. Kline, et al. A privacy-enabled platform for covid-19 applications. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 745–746, 2020.

[41] M. August, C. Davison, M. Diallo, D. Ghosh, P. Gupta, C. Graves, S. Han, M. Holstrom, P. Khargonekar, M. Kline, S. Mehrotra, S. Sharma, N. Venkatasubramanian, G. Wang, and R. Yus. A privacy-enabled platform for covid-19 applications: Poster abstract. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, page 745–746, New York, NY, USA, 2020. Association for Computing Machinery.

[42] P. A. Bernstein and D. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.

[43] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, page 61–71, New York, NY, USA, 1986. Association for Computing Machinery.

[44] G. Blankenagel and R. H. Güting. External segment trees. *Algorithmica*, 12(6):498–532, 1994.

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

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

[47] M. L. Brodie and D. Ridjanovic. On the design and specification of database transactions. In *Readings in Artificial Intelligence and Databases*, pages 185–206. Elsevier, 1989.

[48] P. Carbone et al. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[49] S. Chaudhuri. An overview of query optimization in relational systems. In *17th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1998.

[50] B. Costa, P. F. Pires, and F. C. Delicato. Modeling iot applications with sysml4iot. In *42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2016.

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

[52] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, Oct. 2007.

[53] 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.

[54] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample + seek: Approximating aggregates with distribution precision guarantee. In *SIGMOD Conference*, pages 679–694. ACM, 2016.

[55] 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.

[56] R. Elmasri, G. T. Wuu, and Y.-J. Kim. The time index: An access structure for temporal data. In *16th International Conference on Very Large Data Bases*, 1990.

[57] M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis. Abstract and discrete modeling of spatio-temporal data types. In *Proceedings of the 6th ACM international symposium on Advances in geographic information systems*, pages 131–136, 1998.

[58] M. Erwig, M. Schneider, M. Vazirgiannis, et al. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999.

[59] S. K. Gadia. Toward a multihomogeheous model for a temporal database. In *1986 IEEE Second International Conference on Data Engineering*, pages 390–397. IEEE, 1986.

[60] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In *2013 ACM SIGMOD int. Conf. on Management of data*, pages 1197–1208. ACM, 2013.

[61] D. Ghosh, P. Gupta, S. Mehrotra, and S. Sharma. A case for enrichment in data management systems. *SIGMOD Rec.*, 51(2):38–43, jul 2022.

[62] D. Ghosh, P. Gupta, S. Mehrotra, R. Yus, and Y. Altowim. Jenner: Just-in-time enrichment in query processing. *Proc. VLDB Endow.*, 2022.

[63] S. Giannakopoulou, M. Karpathiotakis, and A. Ailamaki. Cleaning denial constraint violations through relaxation. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, 2020.

[64] M. Goodchild, R. Haining, and S. Wise. Integrating gis and spatial data analysis: problems and possibilities. *International journal of geographical information systems*, 6(5):407–423, 1992.

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

[66] 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.

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

[68] P. Gupta, M. J. Carey, S. Mehrotra, and R. Yus. Smartbench: A benchmark for data management in smart spaces. *Proceedings of the VLDB Endowment*, 13(12), 2020.

[69] R. H. Güting. An introduction to spatial database systems. *the VLDB Journal*, 3(4):357–399, 1994.

[70] A. Haller et al. The modular ssn ontology: A joint w3c and ogc standard specifying the semantics of sensors, observations, sampling, and actuation. *Semantic Web*, 10:9–32, 2018.

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

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

[73] R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4):679 – 688, 2006.

[74] B. Inmon. *Data Lake Architecture: Designing the Data Lake and avoiding the garbage dump*. Technics publications, 2016.

[75] H. Jafarpour, B. Hore, S. Mehrotra, and N. Venkatasubramanian. Subscription subsumption evaluation for content-based publish/subscribe systems. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 62–81. Springer, 2008.

[76] J. Janak and H. Schulzrinne. Framework for rapid prototyping of distributed iot applications powered by webrtc. In *2016 Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 1–7. IEEE, 2016.

[77] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2):111–152, 1984.

[78] J. Jia, C. Li, X. Zhang, C. Li, M. J. Carey, and S. su. Towards interactive analytics and visualization on one billion tweets. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPACIAL '16, New York, NY, USA, 2016. Association for Computing Machinery.

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

[80] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[81] I. Lazaridis and S. Mehrotra. Optimization of multi-version expensive predicates. SIGMOD, 2007.

[82] S. Li, M. Hedley, K. Bengston, D. Humphrey, M. Johnson, and W. Ni. Passive localization of standard wifi devices. *IEEE Systems Journal*, 13(4):3929–3932, 2019.

[83] Y. Lin, D. Jiang, R. Yus, G. Bouloukakis, A. Chio, S. Mehrotra, and N. Venkatasubramanian. LOCATER: cleaning wifi connectivity datasets for semantic localization. *Proc. VLDB Endow.*, 14(3), 2020.

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

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

[86] D. Marmaros, S. E. Whang, and H. Garcia-Molina. Pay-as-you-go entity resolution. *IEEE TKDE*, 2013.

[87] N. May et al. Sap hana–the evolution of an in-memory dbms from pure olap processing towards mixed workloads. *BTW 2017*.

[88] R. McCann, W. Shen, and A. Doan. Matching schemas in online communities: A web 2.0 approach. In *ICDE*, pages 110–119. IEEE Computer Society, 2008.

[89] E. McKenzie and R. Snodgrass. Extending the relational algebra to support transaction time. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 467–478, 1987.

[90] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*. www.cidrdb.org, 2013.

[91] S. Mehrotra, A. Kobsa, N. Venkatasubramanian, and S. R. Rajagopalan. Tippers: A privacy cognizant iot environment. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6, March 2016.

[92] S. Mehrotra, N. Venkatasubramanian, A. Kobas, C. Davison, S. Sharma, R. Yus, G. Bouloukakis, D. Ghosh, P. Gupta, Y. Lin, et al. Privacy cognizant iot environment for the brandeis program. Technical report, UNIVERSITY OF CALIFORNIA, IRVINE, 2022.

[93] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455. ACM, 2013.

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

[95] S. N. Z. Naqvi, S. Yfantidou, and E. Zimányi. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles*, 12, 2017.

[96] M. Nikolic, M. Elseidy, and C. Koch. LINVIEW: incremental view maintenance for complex analytical queries. In *SIGMOD Conference*, pages 253–264. ACM, 2014.

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

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

[99] T. Papenbrock et al. Progressive duplicate detection. *IEEE TKDE*, 2015.

[100] Y. Park et al. Verdictdb: Universalizing approximate query processing. SIGMOD '18.

[101] Y. Park, A. S. Tajik, M. Cafarella, and B. Mozafari. Database learning: Toward a database that becomes smarter every time. In *ACM International Conference on Management of Data*, SIGMOD '17, 2017.

[102] M. Pezzini et al. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. *Gartner (2014, January 28).*

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

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

[105] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. *ACM Sigmod Record*, 25(2):294–305, 1996.

[106] D. M. Powers. Evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation. *J. Mach. Learn. Technol*, 2:2229–3981, 01 2011.

[107] F. Pramudianto, C. A. Kamienski, E. Souto, F. Borelli, L. L. Gomes, D. Sadok, and M. Jarke. Iot link: An internet of things prototyping toolkit. In *IEEE 11th Int. Conf. on Ubiquitous Intelligence and Computing and IEEE 11th Int. Conf. on Autonomic and Trusted Computing and IEEE 14th Int. Conf. on Scalable Computing and Communications and Its Associated Workshops*, 2014.

[108] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[109] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 275–286, New York, NY, USA, 2002. ACM.

[110] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 886–895, April 2007.

[111] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, pages 596–605. IEEE Computer Society, 2007.

[112] 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.

[113] R. Snodgrass and I. Ahn. A taxonomy of time databases. *ACM Sigmod Record*, 14(4):236–246, 1985.

[114] R. Snodgrass et al. Temporal databases. *Computer*, 19(09):35–42, 1986.

[115] R. T. Snodgrass. *The TSQL2 temporal query language*, volume 330. Springer Science & Business Media, 2012.

[116] 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.

[117] K. Stolze. Sql/mm spatial: The standard to manage spatial data in a relational database system. In *BTW 2003–Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW Konferenz*. Gesellschaft für Informatik eV, 2003.

[118] M. Stonebraker and L. A. Rowe. The design of postgres. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, page 340–355, New York, NY, USA, 1986. Association for Computing Machinery.

[119] Y. G. Stoyan, T. Romanova, G. Scheithauer, and A. Krivulya. Covering a polygonal region by rectangles. *Computational Optimization and Applications*, 48(3):675–695, 2011.

[120] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Intermittent query processing. *Proc. VLDB Endow.*, 12(11):1427–1441, 2019.

[121] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Crocodiledb in action: Resource-efficient query execution by exploiting time slackness. *Proc. VLDB Endow.*, 13(12):2937–2940, 2020.

[122] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Thrifty query execution via incrementability. In *SIGMOD Conference*, pages 1241–1256. ACM, 2020.

[123] 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.

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

[125] D. Toman. Point-based temporal extensions of sql and their efficient implementation. In *Temporal databases: research and practice*, pages 211–237. Springer, 1998.

[126] D. Toman. Sql/tp: a temporal extension of sql. In *Constraint Databases*, pages 391–399. Springer, 2000.

[127] A. Toshniwal et al. Storm@twitter. SIGMOD '14.

[128] I. S. Udoh and G. Kotonya. Developing iot applications: challenges and frameworks. *IET Cyber-Physical Systems: Theory & Applications*, 3(2):65–72, 2018.

[129] P. Vassiliadis. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)*, 5(3):1–27, 2009.

[130] S. Wachter. Normative challenges of identification in the internet of things: Privacy, profiling, discrimination, and the gdpr. *Computer law & security review*, 34(3):436–449, 2018.

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

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

[133] Z. Wang, K. Zeng, B. Huang, W. Chen, X. Cui, B. Wang, J. Liu, L. Fan, D. Qu, Z. Hou, et al. Tempura: A general cost based optimizer framework for incremental data processing (extended version). *arXiv preprint arXiv:2009.13631*, 2020.

[134] C. J. Willmott and K. Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.

[135] S. Wu, B. C. Ooi, and K. Tan. Continuous sampling for online aggregation over multiple queries. In *SIGMOD Conference*, pages 651–662. ACM, 2010.

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

[137] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, 2012.

[138] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: generalized online aggregation for interactive analysis on big data. In *SIGMOD Conference*, pages 913–918. ACM, 2015.

[139] X. Zhu and D. Ramanan. Face detection, pose estimation, and landmark localization in the wild. CVPR '12, June 2012.

Appendix A

SmartBench Schema

A.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[...]
}
```

A.2 Devices

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

A.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: {...}
}
```

A.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{..}
}
```