TippersDB: Sensor Observable Data Model for Smart Spaces

Peeyush Gupta[1], Sharad Mehrotra[1], Shantanu Sharma[2], Roberto Yus[3]
[1]UC Irvine, [2]New Jersey Institute of Technology, [3]University of Maryland, Baltimore County

ABSTRACT

This paper presents TippersDB, a database system designed to build sensor-based smart space analytical applications. TippersDB supports a powerful data model that decouples semantic data about the application domain from sensor data using which the semantic data is derived. By supporting mechanisms to map/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. This paper describes TippersDB data model, query-driven translation of sensor data, and provides a summary of the system implementation. The paper further highlights benefits of TippersDB through a case study and performance evaluation of the system using IoT benchmark queries.

1 INTRODUCTION

We explore a new data model and a data management architecture designed to serve as an underlying technology enabler for smart space applications made possible by the emerging sensing and data capture technologies. Today, multiple data management products advertise themselves as platforms for IoT applications [5]. These include standard relational systems, key-value stores, document databases, 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 [1]), stream processing systems (e.g., Storm [35], Spark Streaming [37], Flink [11]), and big data frameworks (such as Hadoop/Hive) that can support complex analytical pipeline 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 paper, in contrast, focuses on what we refer to as the data **virtualization** or the fifth "V" challenge. By 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".

In smart space domains, application logic is best expressed at semantic level, though data arrives at raw sensor level. We call a database system that supports data to be viewed at both levels of abstractions (and provides mechanisms to seamlessly translate concepts, data, queries across them) as supporting data virtualization. We argue that such a database 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. Building applications over large scale sensor deployments require application logic to reason about:

- Which semantic observation can be interpreted from which kind of sensor?
- Given a sensor deployment plan, which sensors can be used to generate which semantic observations based on context such as the location of the sensor, and time (in case the sensor is dynamic)?
- In case multiple sensors offer overlapping capabilities, what criteria should applications use to choose one sensor over another, given the information need?
- How should the application deal with the heterogeneity of sensors that could help detect the same semantic observation? Also, how should application deal with intermittent availability and failures of sensors?

A database technology that supports 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. In addition to the above advantages, 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 paper describes TippersDB, a data management 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 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), 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 [37], Storm [35] – 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 [7, 15]. The alternate strategy of translating sensor data at query time is more suitable. Not only it avoids the large ingestion time delay, 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 does query-driven translation. TippersDB adds translation as an operator inside the query plan, co-optimizing translation, and query processing.

Paper Outline: The paper describes TippersDB data model that supports data virtualization and the design, implementation of TippersDB to implement the model. We begin by first discussing a use case that provides a context and motivation for TippersDB. Then, §3 describes the data and query models of TippersDB. §4 describes TippersDB realization of the data and query models. §5 describes TippersDB translation mechanism. §6 provides a comprehensive evaluation of various design choices and techniques of TippersDB.

2 USECASE: A SMART CAMPUS

We describe a use case of a smart campus both to motivate TippersDB and also to serve as an example to highlight the system features. Such a smart space tracks people's location to support a variety of location-based services. 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. While a smart space would require multiple types of sensors (e.g., HVAC sensors such as temperature, humidity, pressure sensors), for the sake of our example, we focus only on location sensing. 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

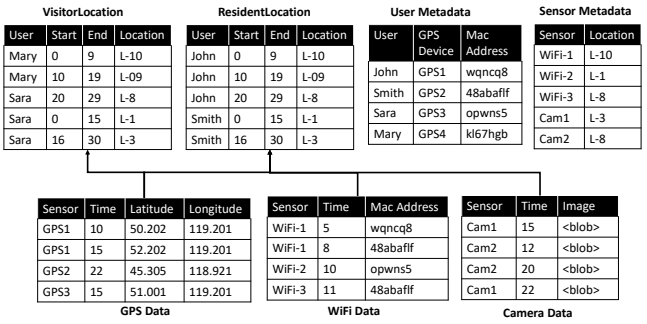


Figure 1: Raw sensor data to Semantic Data.

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. Let us assume that a campus supports passive mechanisms based on WiFi connectivity [22, 23] and cameras (in some parts of the space). 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.

Our goal in this section is to demonstrate the challenges one would face to build the previous applications using existing database technologies and to demonstrate the advantages TippersDB provides. To support location-based applications, a database designer may specify the schema in Figure 1 with tables to store raw sensor data (i.e., GPS, WiFi, and Camera Data). Also, they may create views (or materialized tables) entitled *ResidentLocation* and *VisitorLocation* that represent the location of occupants and visitors of the campus over time, respectively, thereby easing the writing of location applications easier. For example, an application may retrieve the locations (e.g., rooms) and their number of distinct visitors between 9 a.m. and 8 p.m. using the following query.

```
SELECT location, count(distinct user) visits FROM VisitorLocation
WHERE Overlaps([start, end], [8pm, 9pm])
GROUP BY location Order by visits LIMIT 1
```

The logic to convert raw sensory observations into location observations will likely be encoded using user-defined functions (UDFs). For example, consider UDFs *GPS2Location*, *WiFi2Location*, and *Cam2Location* that translate GPS, WiFi, and Camera data, respectively, to generate location information. These UDFs can be used to create the *VisitorLocation* view as follows:

```
CREATE VIEW VisitorLocation AS SELECT GPS2Location(*) FROM
GPSData, UserMetadata UNION ALL SELECT WiFi2Location(*) FROM
WiFiData, UserMetadata, SensorMetadata UNION ALL SELECT
Cam2Location(*) FROM CameraData, UserMetadata, SensorMetadata
```

Such a design faces several complexities:

- ***Multiplicity of Sensors:*** Several different types of sensors could be used to generate the same application-level information. In our example, WiFi connectivity events, GPS, and/or camera all lead to location determination. 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. Programmed as above, the mechanism of how to choose which sensor to use, when, in what context, will have to be part of the application/view definition logic adding significant complexity to the design. TippersDB overcomes such limitations,

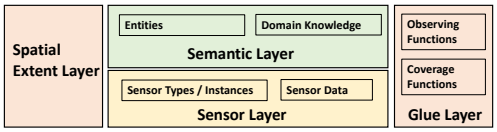


Figure 2: TippersDB data model layers.

since data virtualization supported by its data model allows applications to deal with data at the semantic level, without having to worry about how such data is derived from sensor streams.

• ***Dynamic Sensing:*** 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. 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 using views will make writing code for smart space applications very hard. In TippersDB, new sensor types and new algorithms for deriving semantic data can be added on the fly without modifications to the application code. Also, TippersDB supports dynamic sensors that may join/leave at any time or might be mobile. The spatio-temporal reasoning — to determine if a sensor can/cannot be used to detect a semantic phenomenon of interest based on its spatial and temporal context — is baked into TippersDB, thereby alleviating the need for such logic to be built into applications.

• ***Opportunities for optimization:*** Processing data from different types of sensors can have different execution cost. E.g., for localization, processing one WiFi connectivity event might take ≈20ms, while analyzing a single camera image might take ≈0.4s. 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 thereby minimizing the total execution cost. Adding the burden of optimizing such costs to application logic is complex and, as will become clear later, suboptimal. Instead, significant performance benefits can be gained by co-optimizing such translation costs with query processing as is done in TippersDB. Pushing the logic of translation from application code to database code also allows for several novel optimizations as will be highlighted later.

3 TIPPERSDB DATA MODEL

TippersDB data model is layered and provides an abstraction to write applications independently of the sensor infrastructure (see Figure 2). The *spatial extent layer* models the physical space or the extent of the smart space (§3.1). The *semantic layer* (§3.2) models entities inhabiting the smart space and their relationships. The *sensor layer* (§3.3) models the type and instances of sensors embedded into the space. The *glue layer* (§3.4) provides mechanisms to translate data from the sensor layer into the semantic layer and vice versa.

3.1 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 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 [16, 18, 31] with SQL/MM [31] 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 § 5.1).

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 3 shows the difference between two rectangles (subtracting a 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 3 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 y axis over 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. [20] that considers a more general problem of sequence of differences between d-dimensional spatial objects. We replicate the proof of the bound, as well as, describe our algorithm we use to generate such a rectangular representation.

Theorem 1: n ranges in a one dimensional domain at most create 2n+ 1 non-overlapping ranges over the domain.

Proof: Proof is based on induction. Each range has two bounding points, start and end. Initially the domain consists of only one range.

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**
    $A_1 = Rectangle([B.x_1, B.y_2], [min(A.x_2, B.x_2), A.y_2])$
    $A_2 = Rectangle([A.x_1, A.y_1], [B.x_1, A.y_2])$
    $A_3 = Rectangle([B.x_1, A.y_1], [min(A.x_2, B.x_2), B.y_2])$
    $A_4 = Rectangle([B.x_2, A.y_1], [A.x_2, A.y_2])$
  Return the rectangles from (A_1, A_2, A_3, A_4) with area > 0

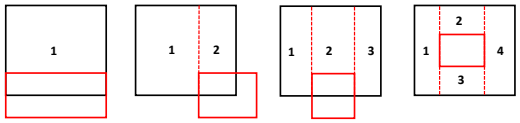


Figure 3: Difference operator.

After adding the first range, the domain will contain three ranges (2*1+1). Assuming the maximum number of partitions resulted from n ranges is 2n + 1, we add the (n + 1)th range. The boundaries of the new range will intersect with at most two of the existing non-overlapping ranges which results in partitioning each of these ranges into two smaller ranges. Therefore, two new ranges is added to the number of partitions and the number of non-overlapping ranges that the domain is partitioned into will be 2n + 1 + 2 = 2(n + 1) + 1 . Using the above proposition, we can provide an upper bound for the number of rectangles (positive and negative) that can be generated by n rectangles.

Theorem 2: Consider a set of n rectangles in d-dimensional space. An upper bound to the number of non-overlapping rectangles that can partition the space based on these rectangles is $O(n^d)$.

Proof: Each of the rectangles has a range in each dimension. Therefore, the domain of each dimension is partitioned with n ranges. The maximum possible number of non-overlapping ranges resulted from this partitioning is 2n+1 in each dimension (according to the Theorem 1). Using the partitioning ranges in each dimension, the d-dimensional space can be partitioned at most into (2n + 1)d non-overlapping rectangles. Therefore, the maximum number of non-overlapping rectangles partitioning the space is $O(n^d)$. Based on the above theorem, the following theorem provides an upper bound for the number of negative rectangles when there are n rectangles.

Theorem 3: Consider a set of n rectangles in d-dimensional space. An upper bound for the number of non-overlapping rectangles that partition the negative space is $O(n^d)$.

Proof: According to the Theorem 2 we know that the upper bound for total number of rectangles partitioning all the space resulted from n rectangles is $O(n^d)$. Each of the given positive rectangles at least include one of the partitioning rectangles. Therefore, the number of remaining rectangles for the negative space is at least $O(n^d n)$ which results in $O(n^d)$.

We observe that despite the n^2 bound, in practice, such a cover includes much less than n^2 rectangles.

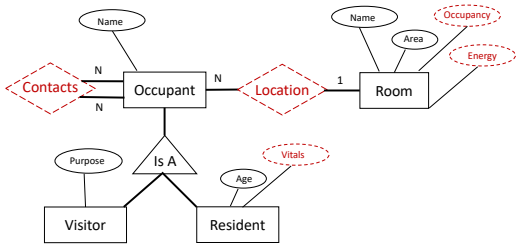


Figure 4: Entities with observable attributes/relationships.

3.2 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.

3.2.1 ***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 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 has 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.

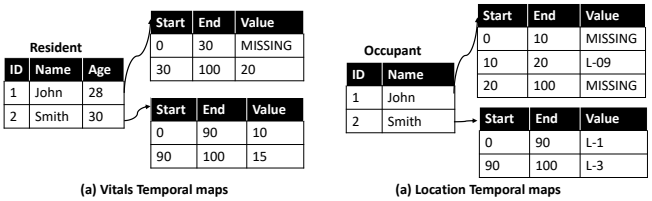
Observable relationship. Relationships between entities in an OER model may themselves be observable - that is, can also be observed using sensor data. For instance, in the OER diagram shown in Figure 4, *contact* is one such relationship since contact between two occupants can be observed using a Bluetooth sensor in their smartphones (it records all Bluetooth devices in close proximity).

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 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.

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);
```

3.2.2 ***Mapping OER Model to Relations.*** Entities and relationships in OER model are mapped to the relations using the standard



(a) Vitals Temporal maps (a) Location Temporal maps

Figure 5: 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 1: Temporal Relations.

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* [33] (that models time as discretized points with a value associated with each time point) or *interval-based* [30] (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:

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

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, *infinity*] and a corresponding `MISSING` value. Figure 5 shows two temporal maps (one for each entity) for 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, 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, 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 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 1c shows the `OccupantContact` temporal relation with flattened multisets.

3.2.3 ***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 [0, 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 OccupantLocation temporal relation (Table 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).

3.3 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);
```

[1] In TippersDB's layered design (discussed in detail in §4), to represent MISSING value, TippersDB reserves a special value for each data type.

```
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 *location* and *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])
```

Aside. Sensors have been modeled in the past literature as devices that observe real-world phenomena and generate observations about measurable properties. The most popular sensor representations are provided by SensorML [10] and the W3C Semantic Sensor Network (SSN) ontology [19]. 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.

3.4 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.

3.4.1 ***Observing Function.*** 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.

3.4.2 ***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 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 w.r.t. 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 §3.3).

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, 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

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

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

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:

$$Coverage^{-1}(p_i, \Gamma, t) = \{(f_l, \{S\}) | \exists \{(p_i, \Gamma_k)\} \in Coverage(f_l, \{S\}, t) \text{ such that } \Gamma_k \cap \Gamma \neq \phi\}$$

As will become clear, 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.

4 TIPPERSDB LAYERED DESIGN

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, [5, 25, 26]). 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. Moreover, a layered implementation can allow identification of potential bottlenecks and challenges, thus, allowing for a more informed and focused native implementation in the future.

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

4.1 Mapping Schema, Data, and Functions

Mapping Space Metadata. TippersDB space model (see §3.1) 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 [3]) (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) are stored together in one table. In our reference implementation, to support very high 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. 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.

4.2 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, who is interested in finding out the occupancy of all rooms that 'John' have visited between time interval [2, 5] could write the query in Figure 6a. The query involves temporal relations for the observable properties location of occupants entity set and occupancy of rooms entity set, i.e., `OccupantLocation` (Table 1b) and `RoomOccupancy` (Table 2a), 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 6a with the corresponding query tree shown in Figure 6b 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 ***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 §5.1 and §4.2.2. 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 6c shows one possible query tree after placing translation operators $\tau_{location}$ and $\tau_{occupancy}$ in the original query tree as shown in Figure 6b.

4.2.1 ***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 ***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 [27]. 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 §5.1.3.

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 7 shows two different valid query trees generated by TippersDB for query

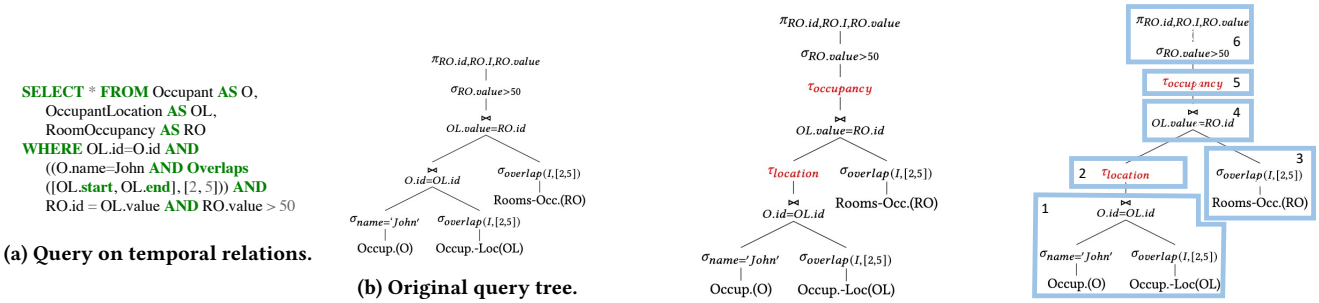


Figure 6: Query processing in TippersDB.

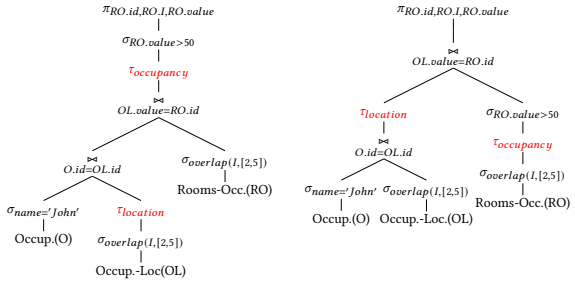


Figure 7: Multiple valid query plans.

tree of Figure 6b. In both the query trees shown in Figure 7, 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 [12] used by the cost-based optimizers.

4.2.2 ***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 6d shows five query blocks created for the query plan shown in Figure 6c. 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 6d 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 executed using a stored procedure that implements the translation operator[4] and takes a temporary relation (output of downstream query block) as input and fills all missing values for a particular observable attribute in it (§5.1 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. Making it non-blocking, however possible, is not straightforward and is out of the scope of this paper.

4.3 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 [23], 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.

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 8) 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 TIPPERSDB TRANSLATION

We describe how the translation operator transforms sensor data to compute `MISSING` values (§5.1). We discuss strategies to further

[4] 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.

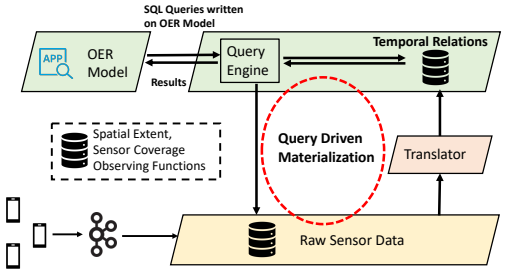


Figure 8: Query-driven materialization architecture.

optimize the translation by exploiting hierarchical data types (§5.2) and by indexing temporal relations for interval search queries (§5.3).

5.1 Translation Operator

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. It 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. Note that 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 `RoomOccupancy` temporal relation (for occupancy property) as shown in Table 2(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 2(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 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.[5] For 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.

5.1.1 ***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

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

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 2: Translation Plan.

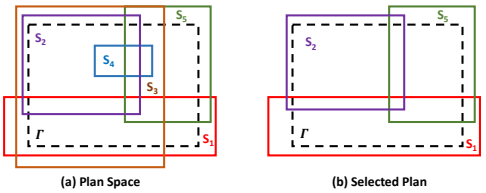


Figure 9: Region-based plan generation.

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.[6]

5.1.2 ***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 are is generated from which a minimum cost plan will be selected.

Plan Space: To generate the plan space, TippersDB calls Coverage^{-1} function (discussed in §3.4.2) on the space Γ_i and a time point t from time interval Δ .[7] 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 overlapping sub-regions inside Γ_i. For example, Figure 9a 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 the problem of covering a polygon with minimum number of rectangles which is NP-complete [32].

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

[6] 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.

[7] 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.

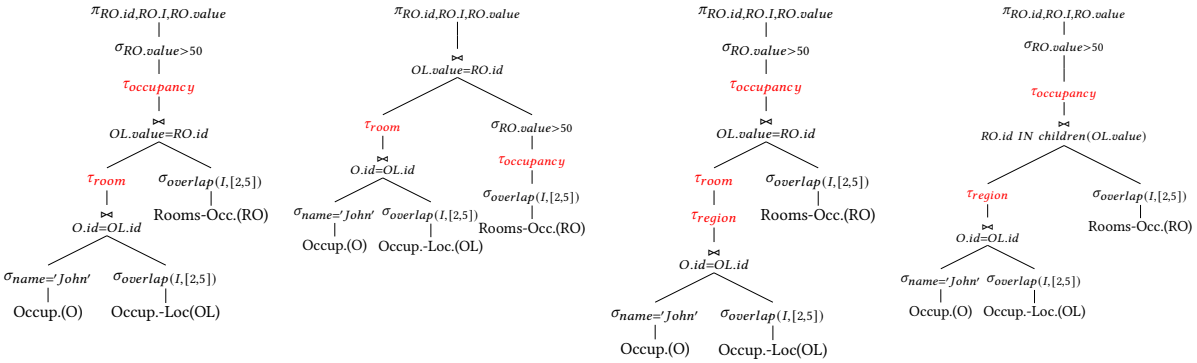


Figure 10: Different query plans with hierarchical translation (Plans 1-4 from Left to Right).

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*(Γ, Δ)
    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, Δ))
  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$_i$ = ϕ
    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*, Γ_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 $S'.area = S'.area - area(\{\Gamma''_{S'}\}), S'.rank = S'.cost/S'.area$

a set of rectangles to be used to cover R that include both 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 §3.1). 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.

5.1.3 *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 Sec 4.2.1). 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 function, and, f^Γ_{min} and f^Γ_{min} are minimum and maximum cost space-based observing function. 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 (§4.2.1).

5.2 Optimizing Hierarchical Data Types

An entity or an observable property in TippersDB can have a hierarchical data type which can be used to reduce the cost of translation by generating more efficient query plans.

Example 5.2 Consider the query given in Figure 6b. 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 10) for the query of Figure 6b.

- **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 p_j op 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 $(parent(p_j)$ op $parent(a_m))$ *AND* $(p_j$ op $a_m)$, 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.

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

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 ≠ L-1" does not imply "region ≠ 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 $parent(a_m)$.

5.3 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 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 [9, 14]. 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 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* (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. 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.

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 3: 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 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 4: Queries.

6 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. Furthermore, in the longer version [**?**], we do experiments to study accuracy of translation operator cost estimation strategy.

6.1 Experimental Setup

For the experiments, we use the smart campus scenario described in §2. We consider that the buildings inside the smart campus are instrumented with WiFi APs, cameras, power meters (WeMo devices [6]), 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 §3.2, the

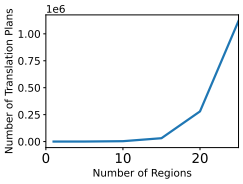


Figure 11: Possible translation plans.

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 12: Sample translation plans.

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 §3.1). 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 [17] to generate synthetic sensor data for a month for a campus with 25 buildings. Table 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 4: Queries Q5-Q8 are extracted from SmartBench [17], 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 3.

6.2 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 w.r.t. 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 11 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 12(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.

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

Table 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 6: Exp 3 - Abstraction overhead.

Eval 3 - Lines of Code: Developing the localization application mentioned in §2 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). Both the codes are listed in Appendix A.

6.3 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 shows the amount of time required to process all the sensor data at ingest using the functions and their cost mentioned in Table 3. For example, Table 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 14 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 §5.2), and b) Indexing of the temporal relation using the strategy in § 5.3 with *interval_threshold*=15 mins. Figure 14 shows that queries Q1, Q4, Q5, and Q8 benefit extensively from the hierarchical optimization - the total query time for Q1 and Q4 reduced 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 14 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 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

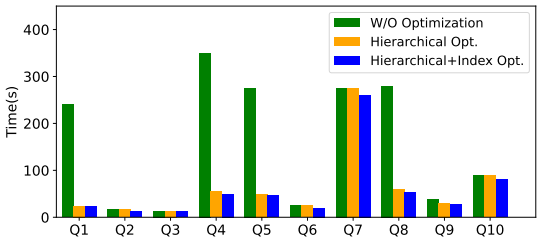


Figure 14: Exp 2 - Performance and effect of optimizations.

result in multiple queries on the underlying database as mentioned in §4.2.2). 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 - 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 13 shows that the total query execution time (and hence the translation cost) of Q1 increases linearly with increasing duration of the time interval.

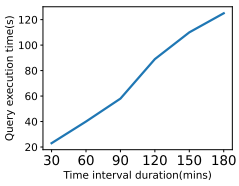


Figure 13: Exp 4 - Effect of query selectivity.

Exp 5 - Accuracy of Translation Operator Cost Estimation In this experiment we study accuracy of translation operator cost estimation technique mentioned in §4.2.1. We measure accuracy of the cost estimation technique by running each query 10 times and calculating the average of the difference (show in Table 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.

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 7: Accuracy of translation operator cost estimation.

7 RELATED WORK

Several **IoT frameworks** [13, 21, 28, 36] have been proposed in the past to ease IoT application development. However, most of these frameworks only abstract out communication and management of IoT devices and do not provide an abstraction over sensors for application developers, who still have to translate sensor data. Similarly, industrial systems like Nest [4], AWS IoT [2] only allow easy access to the sensors and sensor data but fails to provide any semantic level abstraction. **Timeseries DBMSs** [5, 8, 24] are widely used for managing sensor data. These are specialized database systems optimized to store and query a large amount of sensor data. These DBMSs support a very high ingestion rate, compression, columnar storage. These systems, however, are not good to store traditional relational systems, and most of them do not even support full SQL. Several **Temporal DBMSs** [29, 34] were proposed in the past. Temporal databases, along with the current state of the data, maintain historical values (with the time instances when the values were updated). TippersDB uses one of the models proposed for temporal databases to map the OER model to relations. There has been research on **Query Driven Enrichment** in the past. Several papers [7, 15] have shown that query context can be used to eliminate the cleaning of objects that cannot satisfy the query predicates.

8 CONCLUSION

We proposed TippersDB, a database system with a layered data model that provides a semantic abstraction to smart space application developers. TippersDB supports query time translation of sensor data co-optimized with query processing. Our experiments have shown the feasibility of TippersDB in terms of flexibility, extensibility, and performance. While TippersDB is implemented using a layered approach, exploring additional optimizations that are possible at the storage layer and modifying the query processing layer of DBMS are interesting future directions.

REFERENCES

[1] [n.d.]. Apache Kafka. https://kafka.apache.org/23/documentation/streams/.
[2] [n.d.]. AWS IoT. https://aws.amazon.com/iot/.
[3] [n.d.]. GridDB. https://griddb.net/en/.
[4] [n.d.]. Nest. http://www.iot-nest.com/.
[5] [n.d.]. Timescale system. https://www.timescale.com/.
[6] [n.d.]. WEMO. https://www.wemo.com.
[7] Hotham Altwaijry, Sharad Mehrotra, and Dmitri V. Kalashnikov. 2015. QuERy: A Framework for Integrating Entity Resolution with Query Processing. *PVLDB* 9, 3 (2015).
[8] Michael P Andersen and David E Culler. 2016. Btrdb: Optimizing storage system design for timeseries processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*.
[9] Gabriele Blankenagel and Ralf Hartmut Güting. 1994. External segment trees. *Algorithmica* 12, 6 (1994), 498–532.
[10] Mike Botts, George Percivall, Carl Reed, and John Davidson. 2006. OGC sensor web enablement: Overview and high level architecture. In *Int. Conf. on GeoSensor Networks*. 175–190.
[11] Paris Carbone et al. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.
[12] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *17th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*.
[13] Bruno Costa, Paulo F Pires, and Flávia C Delicato. 2016. Modeling iot applications with sysml4iot. In *42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*.
[14] Ramez Elmasri, Gene TJ Wuu, and Yeong-Joon Kim. 1990. The time index: An access structure for temporal data. In *16th International Conference on Very Large Data Bases*.
[15] Stella Giannakopoulou, Manos Karpathiotakis, and Anastasia Ailamaki. 2020. Cleaning Denial Constraint Violations through Relaxation. In *ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*.
[16] Michael Goodchild, Robert Haining, and Stephen Wise. 1992. Integrating GIS and spatial data analysis: problems and possibilities. *International journal of geographical information systems* 6, 5 (1992), 407–423.
[17] Peeyush Gupta, Michael J Carey, Sharad Mehrotra, and Roberto Yus. 2020. Smartbench: A benchmark for data management in smart spaces. *Proceedings of the VLDB Endowment* 13, 12 (2020).
[18] Ralf Hartmut Güting. 1994. An introduction to spatial database systems. *the VLDB Journal* 3, 4 (1994), 357–399.
[19] A. Haller et al. 2018. The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation. *Semantic Web* 10 (2018), 9–32.
[20] Hojjat Jafarpour, Bijit Hore, Sharad Mehrotra, and Nalini Venkatasubramanian. 2008. Subscription subsumption evaluation for content-based publish/subscribe systems. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 62–81.
[21] Jan Janak and Henning Schulzrinne. 2016. Framework for rapid prototyping of distributed IoT applications powered by WebRTC. In *2016 Principles, Systems and Applications of IP Telecommunications (IPTComm)*. IEEE, 1–7.
[22] Shenghong Li, Mark Hedley, Keith Bengston, David Humphrey, Mark Johnson, and Wei Ni. 2019. Passive localization of standard WiFi devices. *IEEE Systems Journal* 13, 4 (2019), 3929–3932.

[23] Yiming Lin, Daokun Jiang, Roberto Yus, Georgios Bouloukakis, Andrew Chio, Sharad Mehrotra, and Nalini Venkatasubramanian. 2020. LOCATER: Cleaning WiFi Connectivity Datasets for Semantic Localization. *Proc. VLDB Endow.* 14, 3 (2020).
[24] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. 2017. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles* 12 (2017).
[25] Yongjoo Park et al. [n.d.]. VerdictDB: Universalizing Approximate Query Processing *(SIGMOD '18).*
[26] Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. 2017. Database Learning: Toward a Database That Becomes Smarter Every Time. In *ACM International Conference on Management of Data (SIGMOD '17).*
[27] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. 1996. Improved histograms for selectivity estimation of range predicates. *ACM Sigmod Record* 25, 2 (1996), 294–305.
[28] Ferry Pramudianto, Carlos Alberto Kamienski, Eduardo Souto, Fabrizio Borelli, Lucas L Gomes, Djamel Sadok, and Matthias Jarke. 2014. 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.*
[29] Richard Snodgrass et al. 1986. Temporal databases. *Computer* 19, 09 (1986), 35–42.
[30] Richard T Snodgrass. 2012. *The TSQL2 temporal query language.* Vol. 330. Springer Science & Business Media.
[31] Knut Stolze. 2003. 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.
[32] Yu G Stoyan, Tatiana Romanova, Guntram Scheithauer, and A Krivulya. 2011. Covering a polygonal region by rectangles. *Computational Optimization and Applications* 48, 3 (2011), 675–695.
[33] David Toman. 1998. Point-based temporal extensions of SQL and their efficient implementation. In *Temporal databases: research and practice.* Springer, 211–237.
[34] David Toman. 2000. SQL/TP: a temporal extension of SQL. In *Constraint Databases.* Springer, 391–399.
[35] Ankit Toshniwal et al. [n.d.]. Storm@Twitter *(SIGMOD '14).*
[36] Itorobong S Udoh and Gerald Kotonya. 2018. Developing IoT applications: challenges and frameworks. *IET Cyber-Physical Systems: Theory & Applications* 3, 2 (2018), 65–72.
[37] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. 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).*

A EXAMPLE

Consider a smart building which is instrumented with different sensors such as cameras, WiFi APs, and bluetooth beacons. Additionally, inhabitants of the building carry smartphones. Let us consider that we define two smart space concepts in TippersDB, Occupant and Room, to represent rooms and inhabitants of the building following the definition in Section **??**. In those concepts, in addition to static attributes, we define the observable attributes "location" of occupants and "occupancy" of rooms. An occupant can be localized using WiFi connectivity data or through bluetooth beacons or through GPS in the phone carried by that user. A room's occupancy can be found using WiFi connectivity or camera data.
Environment Builder: defines the following observation types, sensor types and sensors.

```
CREATE T_ObservationType ConnectivityData(sensorMAC string,
    DeviceMAC Array[string]);
CREATE T_ObservationType GPSdata(lat float, long float);
CREATE T_ObservationType ImageData(blob image);

CREATE T_SensorType WiFiAP(ConnectivityData);
CREATE T_SensorType BluetoothBeacon(ConnectivityData);
CREATE T_SensorType GPS(GPSData);

CREATE T_Sensor WIFIDBH-2065(WiFiAP, T_TemporalMap("ExtLocWifiDBH
      -2065", sde.st_polygon ('polygon ((10.01 20.03, 20.94 21.34,
      35.93 10.04, 10.01 20.03))', NULL, NULL), T_TemporalMap("
      ExtCovWifiDBH-2065", sde.st_polygon ('polygon ((10.01 20.03,
      20.94 21.34, 35.93 10.04, 10.01 20.03))', NULL, NULL))

CREATE T_Sensor BluetoothDBH-2065(WiFiAP, T_TemporalMap("
      ExtLocBTDBH-2065", sde.st_polygon ('polygon ((10.01 20.03,
      20.94 21.34, 35.93 10.04, 10.01 20.03))', NULL, NULL),
      T_TemporalMap("ExtCovBTDBH-2065", sde.st_polygon ('polygon
      ((10.01 20.03, 20.94 21.34, 35.93 10.04, 10.01 20.03))', NULL
      , NULL))

CREATE T_Sensor GPSJohn(GPS, T_TemporalMap<Extent>(),
      T_TemporalMap<Extent>())


CREATE TABLE Room(id int, name char(20), extent T_Extent,
    occupancy int,
    PRIMARY KEY (id));
ADD OBSERVABLE Property TO Room (occupancy int));

INSERT INTO Room VALUES(1, 2072,"Ext2072", sde.st_polygon ('
      polygon ((10.01 20.03, 20.94 21.34, 35.93 10.04, 10.01 20.03)
      )');
```

```
INSERT INTO Room VALUES(2, 2076, "Ext2076", sde.st_polygon ('
     polygon ((10.01 20.03, 20.94 21.34, 35.93 10.04, 10.01 20.03)
     )');

CREATE TABLE Occupant(id int, name char(20), age int,
    OfficeID int,
    PRIMARY KEY (id), FOREIGN KEY (OfficeID)
    REFERENCES Room(id));
ADD OBSERVABLE Property TO Occupant (location int));
ADD OBSERVABLE Property TO Occupant (vitals VitalsData));

INSERT INTO Occupant VALUES(1, John, 25, 1);
INSERT INTO Occupant VALUES(2, Mary, 26, 2);
```

Enrichment Code Developer: defines following observing functions

```
CREATE T_ObservingFunction Acoustic2Location("BLE2Location.java"
     ,{192.0.0.1, 23230, 0.10,0.86}, T_TemporalMap<
     PeopleLocationData>, [{obsType: BLEData, numSensors: 1}]);

CREATE T_ObservingFunction Connectivity2Location("WiFi2Location.
     java",{}, T_TemporalMap<PeopleLocationData>, [{obsType:
     WiFiData, numSensors: 1, sensorSelector: "
     WiFiSensorSelectorC2L", obsSelector: "WiFiObsSelectorC2L"}]);

CREATE T_ObservingFunction GPS2Location("GPS2Location.java",{},
     T_TemporalMap<PeopleLocationData>, [{obsType: GPSData,
     numSensors: 1, sensorSelector: "GPSSensorSelectorC2L",
     obsSelector: "GPSObsSelectorC2L"}]);
```

Queries by Application Developer: Next, the application developer defines and initializes smart space concepts and poses queries required for his application.

```
-- Retrieve list of all people who spent more than 15 minutes with
      Mary on '2020-10-19'

-- Retrieve all the spaces Mary visited that day
WITH A as (SELECT * FROM Occupant as O, OccupantLocation as OL
 WHERE O.name="Mary" AND O.id=OL.id AND overlaps(OL.interval, ['
     2020-10-19 00:00:00','2020-10-19 23:59:59'] ))

-- All the spaces other people visited that day
B as (SELECT * FROM Occupant as O, OccupantLocation as OL
 WHERE O.name!="Mary" AND O.id=OL.id AND overlaps(OL.interval, ['
     2020-10-19 00:00:00', '2020-10-19 23:59:59'] ))

-- People who where in the space as Mary at the same time
C as (SELECT B.name, B.value, intersect(A.interval, B.interval)
FROM A,B WHERE A.value=B.value
AND overlaps(A.interval, B.interval))

-- Those who spent more than 15 minutes in total with Mary
SELECT C.name, SUM(C.interval[1]-C.interval[0]) as ts, B.value
FROM C GROUP BY C.name, C.value HAVING ts > 15
```

In this section, we present the codebase of localization application developed without TippersDB.

```
"""Sensor data processing functions
"""

def gps2location(gpsdata):
    """Takes GPS data as input and returns location of a person"""
    ...
    return location

def bluetooth2location(bledata):
    """Takes GPS data as input and returns location of a person"""
    ...
    return location

def wifi2location(wifidata):
    """Takes WiFi data as input and returns location of a person""
     "
    ...
    return location

"""function to find sensor types"""

def get_sensor_types():
    """return all sensor types"""
    ...
    return [Sensor Types]

"""function to get sensor of a particular types"""

def get_sensor(sensor_type, location):
    """get sensor of a particular type situated at a particular
     location"""
    ...
    return [Sensor]

def get_sensor(sensor_type, owner):
    """get sensor of a partiucal type owned by a particular person
     """
    ...
    return [Sensor]

"""Functions to get data from sensors"""

def fetch_sensor_data(sensor_id, interval):
    """get data from a sensor in a particular interval"""
    ...
    return [(time, payload), ...]

def fetch_sensor_data(sensor_list, interval):
    """get data from all sensors in a list a particular interval""
     "
    ...
    return [(sensor, time, payload), ...]

"""Implementing query that finds out all the people who spent at
     least 15 mins with a particular person
in a particular time interval"""
def execute_query(person, interval):
    sensor_types = get_sensor_types()

    """remove all those sensor types that can not be used for
     localization"""
    for entry in sensor_types:
        if not check(entry, "localization"):
            del sensor_types[entry]

    """Fetch applicable sensors for each sensor type"""
    for entry in sensor_types:
        if entry == "GPS":
            person_gps = get_sensor("GPS", person)
        elif entry == "WiFi":
            wifi_aps = get_sensor("WiFi", "ALL")
        elif entry == "BLE":
            bles = get_sensor("BLE", "ALL")

    """Filter sensor who were down during the time interval"""
    for ap in wifi_aps:
        if not available(ap, interval):
            del wifi_aps[ap]

    """Fetch data from sensors"""
    gps_data =  fetch_sensor_data(person_gps, interval)
    wifi_data =  fetch_sensor_data(wifi_aps, interval)
    ble_data =  fetch_sensor_data(bles, interval)

    """Run sensor processing code to get location information"""
    gps_result = gps2location(gps_data)
    wifi_result = wifi2location(wifi_data)
    ble_result = ble2location(ble_data)

    """Merge the location information from multiple processing
     codes to get final set of locations of the person"""
```

```
    possible_locations = optimize(gps_result, wifi_result,
      ble_result)

    """After finding locations the particular person was in find
      all other people were there at the same location"""

    wifi_aps2 = get_sensor("Wifi", possible_locations)
    bles2 = get_sensor("BLE", possible_locations)

    wifi_data2 = fetch_sensor_data(wifi_aps, interval)
    ble_data2 = fetch_sensor_data(bles, interval)

    """Run sensor processing code to get location information"""
    wifi_result2 = wifi2location(wifi_data2)
    ble_result2 = ble2location(ble_data2)

    person_at_same_location = optimize(wifi_result2, ble_result2)


if __name__ == "__main__":
    execute_query("Mary", (10, 100))
    query_res.show()
```