Sentaur: Sensor Observable Data Model for Smart Spaces

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

ABSTRACT

This paper presents Sentaur, a middleware designed, built, and deployed to support sensor-based smart space analytical applications. Sentaur 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, Sentaur relieves application developers from having to know or reason about either capabilities of sensors or write sensor specific code. This paper describes Sentaur's data model, its translation strategy, and highlights its benefits through real-world case studies.

1 INTRODUCTION

Database systems are widely used in a variety of application contexts as they provide ways to efficiently store/process data and support a data model (e.g. relational data model) that allows easy manipulation and retrieval of data. Database systems provide a general technology that different applications can use to manage their data. However, for application domains having large and complex business logic, directly writing applications on top of a database system using the database's data model becomes challenging. Application developers are well versed with the application logic but, in general, do not have the specialized knowledge of tables and the complex network of relationships between them in the underlying database system.

In the past, application specific middleware systems have been developed on top of existing database systems for several application domains (e.g., Enterprise Resource Planning (ERP) systems [6, 9], Customer Relationship Management (CRM) systems [8], and financial systems [7]). These systems provide higher-level models that capture the essence of the application domain. For instance, a CRM model may have an in-built data model for customer contact, buying history, etc. Such data model is then appropriately mapped, stored and processed on top of a database system by the middleware-based system. This layered approach has been successful and widely adopted since it leverages the power of database systems to manage, store, and process data while supporting higher-level models to make application development easier. Given the importance and emergence of the Internet of Things, sensor-based systems, and smart spaces, we believe it is time to think of specialized middleware to provide appropriate abstractions to build applications on top of sensor data.

In smart space domains, application logic is best expressed at a more semantically meaningful level (i.e., closer to concepts of interest of the application), though data arrives at raw sensor level. We argue that a middleware system that supports data to be viewed at both levels of abstraction (and provides mechanisms to seamlessly translate concepts, data, queries across them) 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?
- Which particular sensors, from the ones deployed in an instrumented space, can be used to generate which semantic observations based on dynamic context (including the location of the sensor and time in case mobile sensors)?
- What criteria should applications use to choose one sensor over another, in case multiple sensors offer overlapping capabilities, 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 applications deal with intermittent availability and failures of sensors?

A system that supports a layered data model 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 thus enabling application writers to write code almost entirely at the semantic level. In addition to the above advantages, such a system is extensible and portable since new sensors and sensor types can be added without modifying application code.

This paper describes Sentaur, a middleware with a data model that makes smart space application development easier by hiding all the complex sensor logic. Sentaur has been deployed in a real-world sensorized environment, a University Campus. The paper describes Sentaur and highlights lessons learnt from its deployment. In summary, Sentaur offers several unique features discussed below:

Semantic Abstraction. Sentaur supports a novel two-tier data model that separates the sensor data from the higher-level semantic data. Sentaur 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. Sentaur 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 their location may change with time. Likewise, relationships between entities may be dynamic. E.g., a person entering a room captured through a sensor may create a new relationship instance between the person and the room. Such dynamic aspects of semantic data that is observed through sensors, are represented using an extended Entity-Relationship (ER), referred to as the Observational ER (OER) model, which represents temporal data evolution. As we will see later, such an OER representation makes data modelling as well as data processing easier.

In addition to representing data at the semantic level, Sentaur 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. Sentaur translates sensor data to generate semantic/application-level information. Sentaur associates *observing functions* with dynamic properties, which observe "value" of the properties through sensors. Also, Sentaur maintains a representation of sensors and their coverage (i.e., what entities in the physical world they can observe) as a function of time.

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 [15, 19]. 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. Sentaur uses such an architecture and does query-driven translation.

Related Works: Sentaur is related to several IoT frameworks [18, 24, 28, 36] proposed in the past to ease IoT application development. These frameworks, however, focus on abstracting communication and management of IoT devices and do not provide an abstraction over sensors for application developers, who still have to translate sensor data. Likewise, industrial systems like Nest [5], AWS IoT [1] only allow easy access to the sensors and sensor data but fail to provide any semantic-level abstraction. Sentaur is also related to Temporal data model supported by temporal databases [29, 34] – Sentaur uses one such model [30] to map the its application-level data model to underlying relational representation.

Paper Outline: First, we describe the data and query models of Sentaur in §2. Then, in §3 we describe the realization of those models on top of a database system. Next, in §4 we describe Sentaur's translation mechanism. §5 describes the deployment of Sentaur in the real world and an evaluation. Finally, §6 discusses future deployments and challenges.

2 SENTAUR DATA MODEL

Sentaur data model is layered and provides an abstraction to write applications independently of the sensor infrastructure (see Figure 1). The *spatial extent layer* models the physical space or the extent of the smart space (§2.2). The *semantic layer* (§2.3) models entities inhabiting the smart space and their relationships. The *sensor layer* (§2.4) models the type and instances of sensors embedded into the space. The *glue layer* (§2.5) provides mechanisms to translate data from the sensor layer into the semantic layer and vice versa.

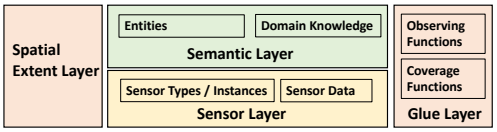

Figure 1: Sentaur data model layers.

Before describing the components of the Sentaur data model, we introduce our deployment scenario to serve as a running example.

2.1 Running Example: A Smart Campus

Consider a smart campus that captures people's location to support location-based services such as: locating group members in real-time, customizing heating, ventilation, and air conditioning (HVAC) controls based on user preferences, contact tracing by 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 campus requires multiple types of sensors (e.g., HVAC sensors such as temperature, humidity, pressure sensors), for the sake of our example, we focus on location sensing. In particular, people can be located based on (a) GPS sensors on their mobile device which transmit the GPS coordinates to the system using a client application, (b) camera-based localization in locations where cameras are installed, (c) using connectivity events in the WiFi network using techniques such as [25, 26]. Each of these mechanisms have their benefits/limitations. GPS-based method works only if the user has actively downloaded an app on their device, and that too, if the user is outdoors. Cameras are typically installed in limited locations and are, furthermore, more time consuming to analyze. WiFi events, on the other hand, provide potentially a ubiquitous localization solution, but might not be as accurate. Mapping application's localization needs requires complex logic to identify and access the set of sensors that individually/jointly meet such needs.

2.2 Spatial Extent Layer

Sentaur 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 [20, 22, 31] with SQL/MM [31] having emerged as a standard to store, retrieve, and process spatial data using SQL. In Sentaur, 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. Sentaur 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, Sentaur supports a variety of spatial predicates (overlap, intersection, meet, etc.) and spatial operations (intersection, union, area, etc.) over spatial objects.

2.3 Semantic Layer

At the semantic level, Sentaur models applications using an extended entity-relation (ER) model that we refer to as ***observable entity-relation*** (OER) model. OER extends the ER model by defining some attributes and relationships to be *observable*.

2.3.1 ***Observable Attributes.*** Observable attributes of entities/relationships are those for which changes can be observed (or computed) using data captured by sensors. Figure 2 shows an example entity set "occupant" that is either a "visitor" or a "resident". The resident entity set has an observable attribute "vitals" whose value changes with time and can be observed, among others, through a smartwatch or Fitbit. Besides observable attributes, an entity may have additional attributes (e.g., occupant entities have an attribute name) the value of which is not associated with any sensor. We refer to such attributes as *n*on-observable or *r*egular attributes.

2.3.2 ***Observable Relationships.*** Relationships between entities in an OER model may themselves be observable - that is, can also be observed using sensor data. For instance, in Figure 2, *contact* is one such relationship since contact between two occupants can be observed, e.g., using the Bluetooth sensor in their smartphones (it records all Bluetooth devices in close proximity).

2.3.3 ***Cardinality Constraints.*** An observable relationship is characterized as *observational 1-1* if an entity e_1 of entity set E_1 at any instance of time can be related to a single entity e_2 of entity set E_2 and likewise, any entity e_2 in E_2 at any time is related to a single entity e_1 in E_1. Note that the definition of observational 1-1 is not identical to that in a regular ER model since an entity e_1 can, indeed be related to one or more entities e_2 and e_3 in the entity set E_2. It is just that e_1 cannot be related to both simultaneously. The concept of observational cardinality constraints generalizes naturally to 1-N, N-1, and N-N relationships. For instance, *L*ocation relationship in Figure 2 is 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 simultaneously be in contact with each other.

2.3.4 ***Participation Constraints.*** An entity set has *total participation* in an observable relationship if every entity of the entity set is related to at least one entity of other entity set at a given time instance. For example occupant entity set has total participation in location relationship since every person is located in some room at a given time instance. An entity set has *partial participation* in an observable relationship if not all entities of the entity set are related to an entity of other entity set at a given time instance. For example room entity set has partial participation in location relationship since there can be rooms with no person located in them.

Sentaur provides the following commands to create entity sets, observable properties, and 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))
```

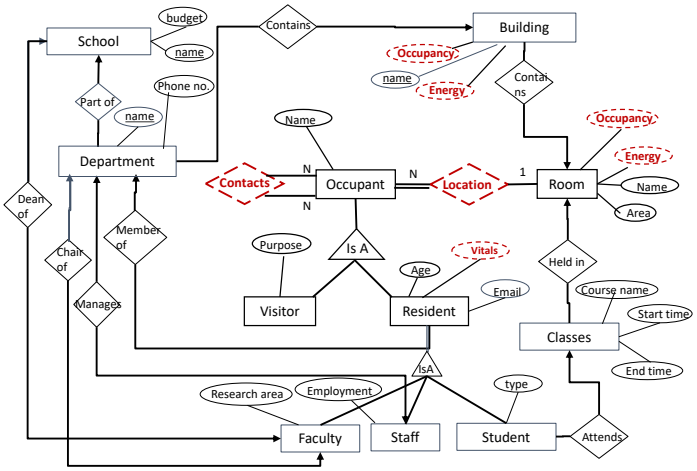

Figure 2: OER model for a smart campus showing entities with observable attributes/relationships (in red).

```
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, N1,TP)
CREATE T_OBSERVABLE RELATIONSHIP Contact(Occupant, Occupant,NN,PP)
```

2.3.5 ***Mapping OER Model to Relations.*** Entities and relationships in OER model are mapped to relations using the standard ER to relational mapping. However, the observable attributes and relationships cannot be modeled as simple attributes as their values change with time. First, we note that prior literature has explored several ways to represent time-varying data. Broadly, techniques are: *point-based* [33] (that model time as discretized points with a value associated with each time point) or *interval-based* [30] (that model time as a continuous timeline with a value associated with each time interval). Sentaur 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, Sentaur 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 3 shows two temporal maps (one for each entity) for vitals observable

[1] In Sentaur's layered design (discussed in detail in §3), to represent MISSING value, Sentaur reserves a special value for each data type.

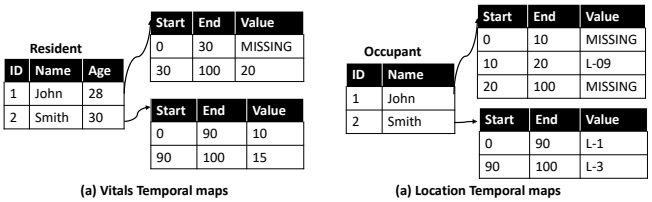

Figure 3: Mapping OER to Temporal Maps.

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.

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 2, 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 2, 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.

2.3.6 ***SQL with Temporal Relations.*** Sentaur 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], [0, 15])
```

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 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).

2.4 Sensor Layer

At the sensor layer, Sentaur provides a way to specify *sensor type*, to associate *observation type*, and to instantiate sensors in the system. As an example, the following first three commands define two observation types: `ConnectivityData` (i.e., a data type including device and AP mac address), `ImageData` and `VideoData`, and the last two commands define two sensor types: `WiFiAP` that generate `ConnectivityData` and `Camera` `ImageData` and `VideoData` observation types.

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

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

After defining observation and sensor types, sensors can be instantiated in the system. Sensors in Sentaur are classified as: (1) *space-based* that generate observations in a physical region and are referred to as covering that space (irrespective of the entity they observe) or (2) *entity-based* that generate observations 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 sensor. 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>);
```

where *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 Sentaur, 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)
```

where `entity` refers to the entity that this particular 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, Sentaur 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 [17] and the W3C Semantic Sensor Network (SSN) ontology [23]. 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.

2.5 Glue Layer

The glue layer in Sentaur 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 which data from which sensors can be used to generate which semantic observations.

2.5.1 ***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 the function may take and the observing attribute it 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.

2.5.2 ***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 §2.4).

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 Sentaur, however in case it is not specified, Sentaur uses the physical coverage of a sensor as its semantic coverage.

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

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

As will become clear, Coverage^{-1} is computed, when processing any query that contains observable attributes or relationships.

3 MAPPING SENTAUR TO DATABASE SYSTEM

In this section, we describe how Sentaur schema, data, functions, and queries are mapped/layered on the underlying database system.

3.1 Mapping Schema, Data, and Functions

Mapping Space Metadata. Sentaur space model defines the hierarchical spatial extent including geographical bounds of buildings, regions, and rooms. Sentaur creates special tables called `Spatial Metadata` in the underlying database to store spatial metadata.

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 [4]) (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

[2] This work does not deal with actuators that perform actions (e.g., switching something on/off).
[3] Currently Sentaur supports Python and Java-based functions.

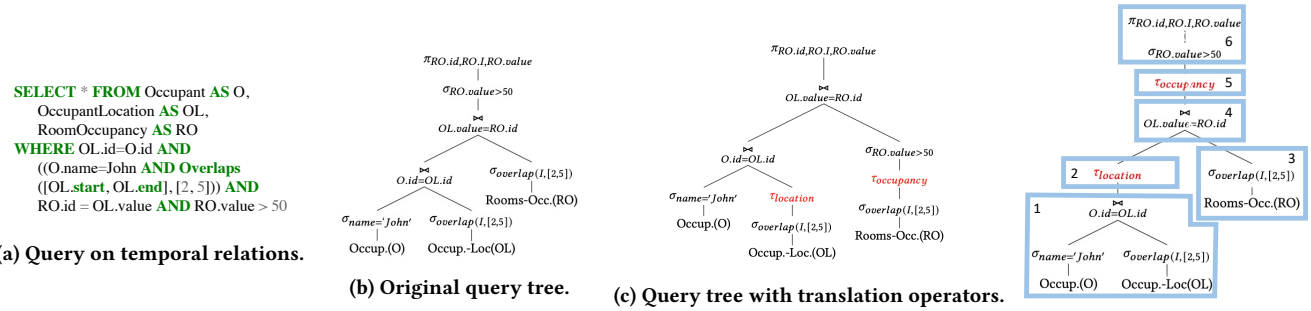

Figure 4: Query processing in Sentaur.

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 (see [3]), to support very high data rates, observations are first pushed to a message queue before they are stored in the database system.

Mapping Functions. Sentaur currently supports Python and Java-based observing and coverage functions. Sentaur 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.

3.2 Mapping Queries

In Sentaur, 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 2, 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 4a. 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 4a with the corresponding query tree shown in Figure 4b cannot be executed directly on the underlying database. To fill the missing values during query execution, Sentaur 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 §4.1 and §3.2.1. Sentaur updates the original query plan by placing translation operators above the selection predicate on every temporal relation present 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 4c shows one possible query tree after placing translation operators $\tau_{location}$ and $\tau_{occupancy}$ in the original query tree as shown in Figure 4b.

3.2.1 ***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, Sentaur divides it into query blocks such that a block contains either only translation operators or only relational operators. Figure 4d shows five query blocks created for the query plan shown in Figure 4c. After creating the query blocks, Sentaur 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 4d 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 (§4.1 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 SENTAUR TRANSLATION

In this section we describe how the translation operator transforms sensor data to compute `MISSING` values.

4.1 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 4.1 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

[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, Sentaur implements the translation operator as a stored procedure.

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.

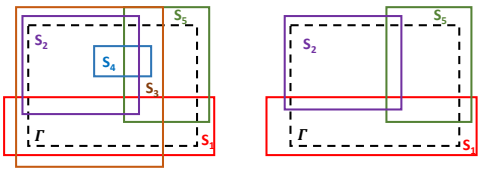

Figure 5: Region-based plan generation.

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 Sentaur. 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). ■

Since the coverage of sensors can change with time, Sentaur first 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, Sentaur generates two types of plans: entity-based and region-based plans; and of the plans generated, it selects the one with a lowest cost.

4.1.1 ***Entity-Based Plan*** . Recall that entity-based sensors always observe a particular entity irrespective of the space they are in. To generate an entity-based plan, Sentaur, for each entity in the input to the translate operator, finds the minimum cost observing function and the entity-based sensor that can observe the given entity for the entire Δ. We generate this plan only when the entities are explicitly listed on the input, i.e., not ALL, or if the region-based plan is not feasible.[6]

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

Plan Space: To generate the plan space, Sentaur calls Coverage^{-1} function (discussed in §2.5.2) 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 5a represents one such plan space and shows a region Γ with different sets of sensors covering different overlapping sub-regions of Γ.

Plan Selection: Executing all the observing functions (with the corresponding sensors) included in the plan space will result in redundant translation work, since multiple sets of sensors might cover overlapping sub-regions of the given region. Sentaur 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 a set of rectangles which is NP-complete [32]. 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 i.e., 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 $\Gamma_{uncovered}$. 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 SENTAUR DEPLOYMENT AND EVALUATION

We have deployed Sentaur at the UC Irvine campus. Sentaur collects WiFi connectivity data from access points in more than 30 buildings, as well as, camera data, WeMo devices [14], and HVAC sensors instrumented in the Computer Science buildings, as well as, GPS data from smartphones of a few users willing to share the data in order to support variety of services. Sentaur collects more than 31 million events per day with data rates as high as 5,000 events/second during peak hours. Sentaur abstracts the collected sensor data into semantically meaningful information to support a variety of applications/services. These include locating individuals/groups in real-time, finding most frequently met persons and most frequently

[5] If the coverage changes (e.g., a sensor becomes available/unavailable during a Δ interval), then Sentaur 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.

[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.

visited locations, monitoring occupancy of different parts of a building and analyzing building usage over time, finding energy usage of different regions of a building over time, analyzing correlation of between faculty-student interaction, course attendance to grades. The most extensively used primary applications supported by Sentaur create awareness during COVID-19 pandemic. In particular, they empower: 1) Campus administrators to make informed decisions based on dynamic building occupancy; 2) Individuals to decide about visiting (parts of) a building by monitoring the number of unique visitors that have passed through a given region; and 3) Individuals to monitor if they could have been exposed to COVID-19 by being in locations reported as infectious by UCI. Sentaur's support for rapid application prototyping enabled us to quickly develop such apps which were used operationally for over year for this purpose [13]. Each of these applications can be seen showing information in real time at [10].

5.1 Impact & Benefits

To evaluate the benefits of Sentaur for smart space application developers, we consider the essential functionality required by most of the applications in our deployment: inferring the location of a person during a given time interval.

Eval 1 - Lines of Code: We evaluate first the benefit of Sentaur in simplifying application development. To this end, we wrote the previous functionality with Sentaur and without it (i.e., directly on sensor data). In Sentaur this functionality corresponded to a simple SQL query *"SELECT * FROM Occupant O, OccupantLocation OL WHERE O.name='?' AND OL.id=O.id AND Overlaps ([start,end], [10,50])"* . In contrast, coding this directly over sensor data required over 500 lines of code,[8] and that code, while it supported multi-sensor reasoning, was still not fault-tolerant when sensors could fail/change. This example illustrates clearly the power provided by Sentaur in building sensor-based applications.

Eval 2 - Extensibility: Porting application functionality to different smart spaces or conditions might require completely different code depending on the available sensors/observing functions. Sentaur hides such complexities from developers by dynamically adapting to the available resources. A developer using Sentaur does not have to change the application code beyond making simple parameter changes in their queries. We evaluate the benefit of Sentaur in supporting extensibility and portability. Particularly, we consider a small change in the time range parameter of the running example query/functionality. Version 1 and 2 of the query choose different time ranges ((50, 200) and (100, 250), respectively). Such a small change in the query results in totally different translation plans as shown in Figure 7 (generated to fill the missing values in `OccupantLocation` temporal relation). Without Sentaur, the application developer would have to write the code required for each plan involved in variations of the same functionality.

Eval 3 - Translation Plan Space Complexity: Sentaur 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 applications.

The number of plans considered by Sentaur is an indicator of the simplification the system offers - without using Sentaur the developer would need to reason about such plans. Figure 6 shows the number of possible translation plans for the sample query involving a building with 25 regions. Note that there are more than 1 million possible plans in this case which increase exponentially with the increase in the number of regions to be covered.

[8]This code is listed in Appendix A.

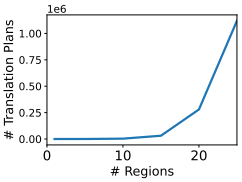

Figure 6: Possible translation plans.

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 7: Sample translation plans.

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.

5.2 Performance Evaluation

The previous section evaluated the feasibility of Sentaur wrt to its main design goal: facilitating application development. Another motivation for Sentaur, even if it was not the main goal, was to improve performance of applications by exploring query-time translation. Before, deploying Sentaur, we built a subset of the applications using a traditional approach based on abstracting sensor data at insertion time. This approach was not able to sustain data rates at peak hours (in the order of 5,000 events/second). Processing, for instance, WiFi events to generate localization information using a machine learning method [26] took more than 200 seconds for data collected in just 1 second. With Sentaur's deployment, using its query time translation, the data ingestion is now almost instantaneous, and data is available for analysis without any delay.

In the following, we evaluate the performance and scalability of Sentaur in our deployment. For reproducibility purposes and to provide performance results on standardized data, we use synthetic sensor data for this experiment since the real sensor data captured in the campus cannot be shared. We generate synthetic sensor data, for a campus with 25 buildings for a month, using the sensor data generation tool provided in the IoT database benchmark SmartBench [21]. Table 3 shows the number of instances of each sensor type, the number of rows, and the size of the generated sensor data.

Experimental Setup. We select ten representative queries ranging from simple selection queries (e.g., to find the location of a person or occupancy of a room in a given time interval) to more complex queries involving multiple joins and aggregations (e.g., to find all "health" individuals who visited locations in which an "unhealthy" person had been before or to find the average time spent by users in different types of rooms). Detailed description of each query is give in Table 4. All queries are expressed on the semantic model referring

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.

to entities and their observable (or not) attributes. Note that during the evaluation we set the value of the time interval parameter to be 30 minutes unless explicitly stated. We use observing functions from our deployment 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. 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 as in our deployment. We note that we could use Timeseries DBMSs [12, 16, 27] to manage sensor data to further improve Sentaur's performance.

Exp 1 - Eager Evaluation: As we mentioned before, processing/-translating the entire sensor data to generate meaningful observations as it arrives is not practical. Table 5 shows the time required to process sensor data at ingest using the functions with cost mentioned in Table 3. The results show that processing one month of sensor data requires ≈141 days, which is not practical.

Exp 2 - Query Processing in Sentaur: Table 6 plots query execution time using Sentaur with a database in a cold start (i.e., none of the data has been processed using any of the observing functions). Query execution times range from 13s to ≈6 minutes (for a complex query to retrieve all rooms with occupancy above a threshold). Table 6 further shows the percentage of the total time spent in translation, generating translation plans, and executing queries on the underlying database.[9] The time spent in running observing functions (i.e., "Translation" in Table 6) is computed by subtracting the rest of the costs from the total time. Note that executing observing functions consumes 70%-95% of Sentaur execution time while planning and executing queries in the underlying database takes a small fraction of it. This indicates that queries will become significantly faster when queries are executed over data that has been partially processed using observing functions, since the dominant cost of execution is that of executing such functions.

[9]Executing a query in Sentaur may result in multiple queries on the underlying database as mentioned in §3.2.1.

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
Total Time	240s	17s	13s	350s	275s	25s	275s	280s	38s	89s
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 2 - Cost breakdown for queries in Sentaur.

Exp 3 - Effect of Selectivity: We study the effect of the query selectivity on Sentaur's query execution time. Figure 8 shows execution time for one of the queries (Q1 in Table 6) when we increase its time interval parameter. The execution time (and hence the translation cost) increases linearly with increasing duration of the time interval. This is expected given that the overhead of translation is minimal and the dominant cost is that of executing observing functions.

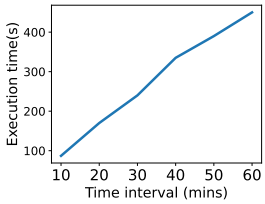

Figure 8: Exp 3 - Effect of query selectivity.

6 FUTURE DEPLOYMENTS AND DISCUSSION

We next discuss two early stage deployments of Sentaur in the domains of assisted living and wildfire resilience. The assisted living smart space setting, which is a part of the CareDEX project [2], aims to ensure safety of older adults who require personalized care. Here, senior housing facilities and residents are instrumented with diverse sensors (fall detection, wander alerts, motion detectors) enabling personalized monitoring of residents by caregivers to identify those in need of urgent care. Information generated from in-situ motion sensors and WiFi access points are merged with mobile fall detection or wander alert sensors in Sentaur to create improved awareness applications for caregivers. Sentaur applications will support analysis to pinpoint unsafe regions where most falls have taken place, identify isolated residents who exhibit low levels of interaction with others, and track residents who are an elopement risk and at danger of leaving the facility. OER model for an assisted living scenario is given in Figure 9. We have already created some prototype dashboards for the deployments. Figure 11 shows as spatial region of an assisted living space modeled using Sentaur, Figure 12 shows coverage of different WiFi access points in the space and Figure 13 shows a dashboard running several queries on the OER model.

Another novel use of Sentaur is in the context of *Prescribed Fire Monitoring* in the SPARx-Cal project [11]. Prescribed Fires are planned, intentional and controlled applications of fire to a land area, under specified predicted weather conditions - it is used as a proactive technique to prevent rapid spread of wildfires in forests and wild-land urban communities. Prescribed fires run the risk of escaping; fast and accurate monitoring of their progress is critical. In SPARx, a range of devices/sensors are used to monitor burn progress, drones are used to capture aerial imagery of fire levels

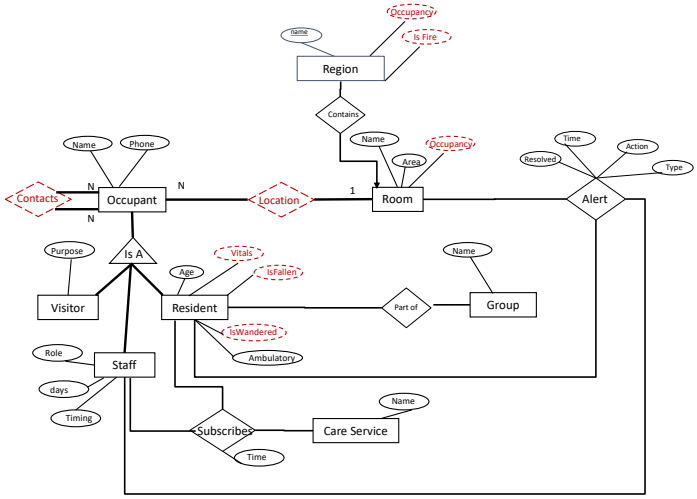

Figure 9: OER model for an assisted living space showing entities with observable attributes/relationships (in red).

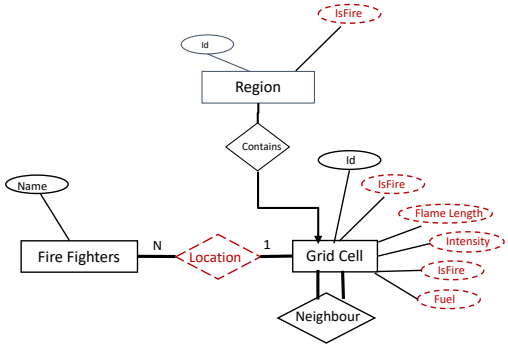

Figure 10: OER model for a prescribed fire exercise showing entities with observable attributes/relationships (in red).

while insitu sensors at the burn site capture environmental parameters (wind, smoke, air quality). Thermal and RGB images from drones and insitu cameras are analyzed for fire presence, flame length and fire intensity in different regions. Simultaneously, wind sensors (wind direction and speed), air quality sensors (levels of particulate matter and smoke) and humidity sensors provide local environmental conditions that dictate fire progress. GPS devices carried by personnel are used for crew location to direct them to regions where attention is required for ignition or control. Through a Sentaur deployment, an analyst will be able to pose queries to better monitor the burn (identify regions with fire presence/absence, determine regions with unfavorable wind/humidity conditions, locations of fuel with low humidity) and control further data collection through drone path planning. OER model for a prescribed fire scenario is given in Figure **??**.

We expect Sentaur to provide similar benefits in these contexts as in the UCI smart space deployment. It will enable us to develop these use cases by writing applications at the semantic level, leaving complex sensor translation logic, dealing with sensor heterogeneity and multiplicity, as well as ways to tolerate sensor failures, to the

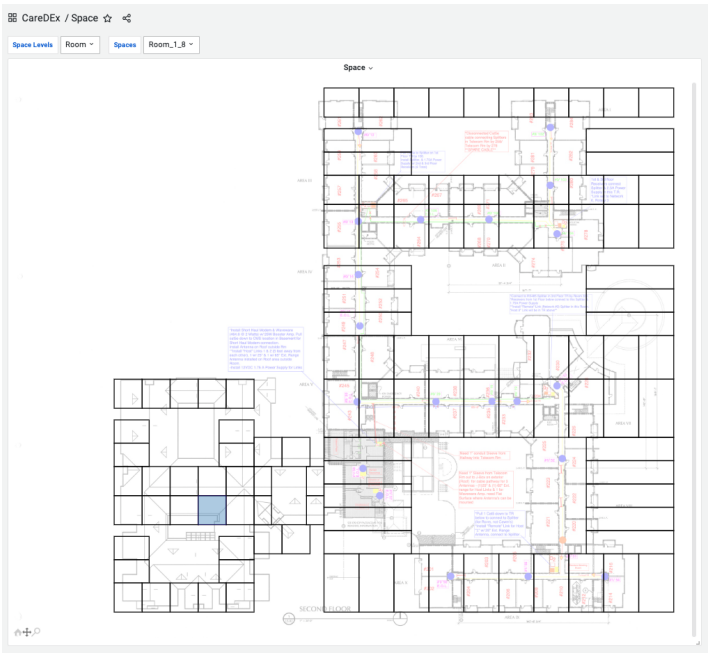

Figure 11: Screenshot showing space model of a assisted living space.

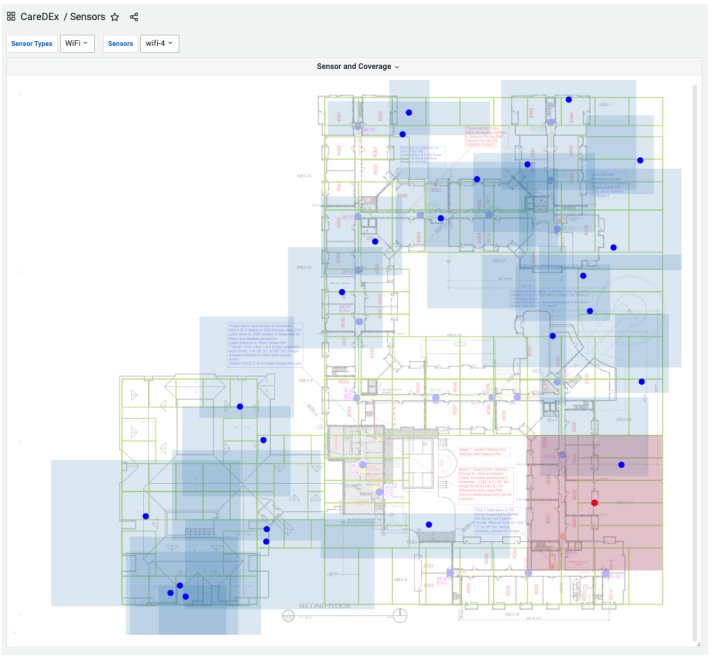

Figure 12: Screenshot showing coverage of WiFi access points in an assisted living space.

Sentaur middleware. Nonetheless, the two new application contexts pose several new challenges that we anticipate having to address. In Sentaur, query-time translation mitigates the large ingestion delay

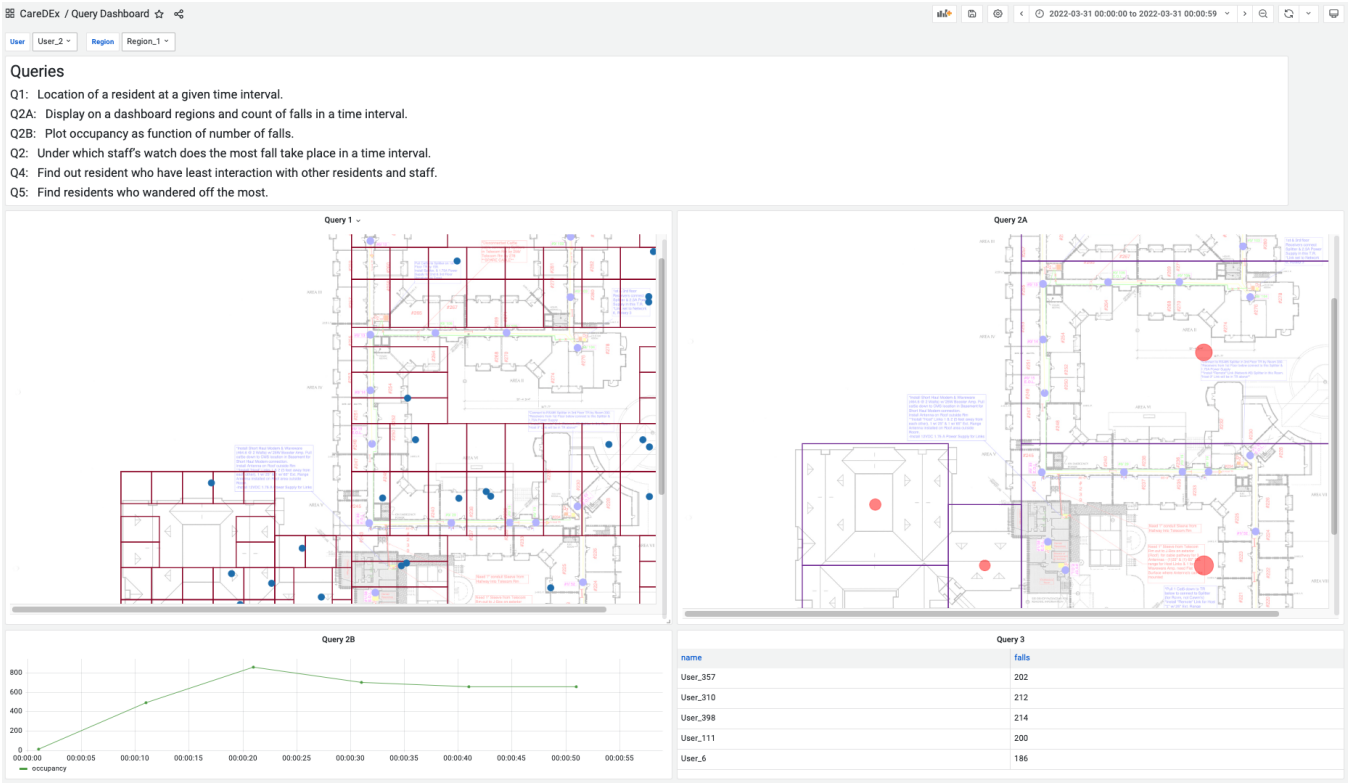

Figure 13: Screenshot showing a dashboard running various queries in the assisted living space scenario.

but it can cause queries to take a long time to complete, specially during the cold start phase. Increased query latency could result in unacceptable application performance, especially when query results drive real-time control as in drone planning the prescribed fire domain. We will need to explore variety of optimizations to reduce the translation cost and therefore the query execution time. For example, we can exploit the query semantics and place translation operators optimally to remove redundant translations.

REFERENCES

[1] [Online; accessed May-2022]. *AWS IoT.* https://aws.amazon.com/iot/

[2] [Online; accessed May-2022]. *Enabling Disaster Resilience in Aging Communities via a Secure Data Exchange.* https://sites.uci.edu/caredex/

[3] [Online; accessed May-2022]. *Extended version.* https://github.com/ucisharadlab/tdb

[4] [Online; accessed May-2022]. *GridDB.* https://griddb.net/en/

[5] [Online; accessed May-2022]. *Nest.* http://www.iot-nest.com/

[6] [Online; accessed May-2022]. *Oracle ERP.* https://www.oracle.com/erp/

[7] [Online; accessed May-2022]. *Oracle Financials.* https://www.oracle.com/erp/financials-cloud/

[8] [Online; accessed May-2022]. *Salesforce CRM.* https://www.salesforce.com/crm/

[9] [Online; accessed May-2022]. *SAP S4/Hana ERP.* https://www.sap.com/products/s4hana-erp.html

[10] [Online; accessed May-2022]. *Sentaur Applications.* https://tippersweb.ics.uci.edu/covid19/d/IwAc1O9Wk/covid-19-effort-at-uc-irvine?orgId=1

[11] [Online; accessed May-2022]. SPARx Cal: Smart Practices and Architectures for Rx fire in California. https://sites.uci.edu/sparxcal/

[12] [Online; accessed May-2022]. *Timescale system.* https://www.timescale.com/

[13] [Online; accessed May-2022]. *UCI Uses Campus Wi-Fi to Test COVID-19 Contact Tracing App.* https://healthitanalytics.com/news/uci-uses-campus-wi-fi-to-test-covid-19-contact-tracing-app

[14] [Online; accessed May-2022]. *WEMO.* https://www.wemo.com

[15] Hotham Altwaijry, Sharad Mehrotra, and Dmitri V. Kalashnikov. 2015. QuERy: A Framework for Integrating Entity Resolution with Query Processing. *PVLDB* 9, 3 (2015).

[16] 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).*

[17] 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.

[18] 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).*

[19] 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).*

[20] 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.

[21] 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).

[22] Ralf Hartmut Güting. 1994. An introduction to spatial database systems. *the VLDB Journal* 3, 4 (1994), 357–399.

[23] 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.

[24] 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.

[25] 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.

[26] 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).

[27] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. 2017. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles* 12 (2017).
[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 APPLICATION DEVELOPMENT WITHOUT SENTAUR

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()
```