Design of an Effective Ontology and Query Processor Enabling Portable Building Applications

by

Gabriel Tomas Fierro

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science, Plan II

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David E. Culler, Research Advisor
Professor Randy H. Katz

Spring 2019

The thesis of Gabriel Tomas Fierro, titled Design of an Effective Ontology and Query Processor Enabling Portable Building Applications, is approved:

Research Advisor	______________________________	Date	__________
	______________________________	Date	__________

University of California, Berkeley

Design of an Effective Ontology and Query Processor Enabling Portable Building Applications

Copyright 2019
by
Gabriel Tomas Fierro

Abstract

Design of an Effective Ontology and Query Processor Enabling Portable Building Applications

by

Gabriel Tomas Fierro

Master of Science, Plan II in Computer Science

University of California, Berkeley

Professor David E. Culler, Research Advisor

Buildings have long been the target of applications seeking to reduce energy consumption, increase occupant productivity and comfort and improve building/grid operation. However, these advances rarely see widespread adoption due to the prohibitive cost of implementing these applications on each building. This cost arises from the fact that most buildings are highly customized and have no machine-readable description of their structure or the systems involved in their operation. We propose that a flexible, expressive schema describing the structure and process of a building and its subsystems can enable the mass-customization of energy efficiency applications.

This thesis presents the design of Brick, a graph-based metadata schema for buildings that captures the entities ("things") in a building and the relationships between them. We demonstrate how Brick's extensible class hierarchy is able to define the sets of entities required by energy efficiency applications by using Brick to model a suite of real-world buildings. Applications execute against Brick models by querying them for the information they need to operate. Queries are expressed using Brick's relationships, which capture associations between entities such as composition, influence, measurement and location. Together, these features of Brick enable an expressive, standardized, digital representation of buildings.

We demonstrate how the Brick schema is implemented with the RDF data model and how models of buildings are queried with the standard SPARQL query language. This informs an investigation of the systems requirements for the infrastructure storing models and processing queries against them, involving a description of the expected Brick workload and an evaluation of existing RDF/SPARQL technologies. We then design and implement a performant query processor – HodDB – that provides interactive-level query latencies (sub 100ms). We evaluate HodDB on a synthetic Brick workload and demonstrate how it is used to implement novel integrations of Brick with data analysis and control systems.

Contents

List of Figures

List of Tables

Acknowledgments

Chapter 1

Introduction

The U.S. Department of Energy reports that commercial buildings constituted 47% of all energy consumed in the U.S. in 2017 [107]. On average, 30% of this energy is wasted [34]. The increasing availability of digital monitoring and control systems presents a tremendous opportunity to reduce building energy consumption, increase occupant productivity and comfort, and improve building/grid operation. There has been substantial interest in these areas both from academia – including work on occupancy-driven control [2, 3, 56], automated fault detection and diagnosis [90, 81, 53, 64, 114] and model-predictive control [44, 75, 93] – and industry – from grid providers and building management and control vendors to energy service companies (ESCOs) and ancillary services [29, 18, 35, 94, 55].

The most significant barrier to the adoption of energy efficiency applications is cost [30]. Buildings differ not only in their architecture and usage but also in the structure and composition of the internal systems and processes involved in their operation such as lighting, power, conditioned air, security and fire protection. As a result, a major factor in these costs is the time and effort required to customize the operation of an application suite to the specific, and often idiosyncratic, configuration of spaces, equipment, sensors, controllers and other components and software interfaces present in a particular building. Reducing these costs means enabling the mass customization of building applications to the existing building stock. This requires normalizing descriptions of buildings and their digital resources to a standard scheme. The primary difficulty with such large scale conversion is the heterogeneity and availability of this information: digital representations of buildings rarely capture all the relevant entities and relationships necessary for building portable applications, and what information is exposed does not usually follow a sufficiently descriptive naming scheme.

As a result, the customization an application to multiple buildings is a largely manual process due to the lack of sufficiently descriptive *metadata* that captures the relevant information required to implement an application. This metadata consists of a description of the “things” (*entities*) in a building – be they logical, virtual or physical – and the *relationships* between those things – how they are connected, contained, used, located and behave. An effective metadata schema for describing buildings will allow applications to be *portable* – “write once, run anywhere” – and significantly reduce the costs of developing new applications and adapting existing applications for buildings at scale. The metadata schema must be usable in real-world settings; given the complexity and size of modern building management systems, the corresponding digital representation of the building should enable tools and interfaces that ultimately reduce the cost of implementing portable applications at scale.

This thesis presents the design and implementation of a metadata schema for buildings: a machine-readable representation of the equipment, sensors, actuators, assets, systems and spaces present in a building and the relationships between them. Chapter 2 presents the context for this work: it reviews the state of the art in digital control systems for large commercial buildings, and presents an overview of the metadata solutions that describe those systems. Existing metadata solutions are insufficient for supporting the development of portable building applications because they are incomplete, inexpressive and informal. This motivates the need for a new, standardized metadata schema that is able to describe the structure and composition of buildings and the processes involved in their operation.

Chapter 3 presents the design of Brick, a graph-based building metadata schema that represents the entities and relationships necessary for implementing portable applications. Brick defines an extensible class hierarchy that describes entities across a wide array of building subsystems. Entities are associated with one another using a small number of expressive relationships defined by Brick, capturing concepts like measurement, composition and sequence. Brick advances the state-of-the-art by formalizing the semantics of its data model, which enable expressive power with clean extensibility necessary for the construction of portable applications. Chapter 4 presents the implementation and formalization of Brick as an ontology using the RDF data model. The RDF data model defines a directed, labeled graph that can be queried using the SPARQL query language. This abstraction sufficiently describes the Brick model, and the use of formal ontologies enables the expressiveness and extensibility of Brick.

Chapter 5 presents a methodology for the synthesis of Brick models from existing sources of metadata and demonstrates the conversion for several common sources of metadata including common semi-structured human-readable labels, computer-aided design models for construction and other existing standards for the digital representation of buildings.

Chapter 6 evaluates the efficacy of Brick in enabling the implementation of applications that generalize across multiple buildings. In a case study of six Brick models representing six real-world buildings, Brick is able to describe upwards of 98% of the information available in each building's digital management system. The chapter discusses the implementations of eight representative building applications drawn from the literature which are executed against each of the six Brick models. This evaluation demonstrates that Brick is able to describe all salient entities and relationships required by these applications, and that the applications are able to generalize despite the diversity of the case study buildings.

Chapter 7 examines a sample Brick application workload to determine the latency and expressivity requirements of a query processor for Brick. Brick applications require a low-latency query processor and rely upon a specific subset of SPARQL language features. These workload properties present performance issues that are pathological to the design of existing RDF/SPARQL query processors. Chapter 8 presents the design and implementation of a performant Brick query processor – HodDB – that takes advantage of the properties of the Brick workload. This enables interactive systems that facilitate the development of portable applcations as well as novel tools for data exploration and analysis.

Chapter 9 covers future work, discusses current industrial collaborations around Brick and concludes.

Chapter 2

Background and Prior Work

Buildings are characterized by extreme heterogeneity. Each building has a unique structure and composition of systems tailored to the particular environment and uses of the building. Deploying an energy efficiency application on a building means implementing a custom application that accounts for the specific, and often idiosyncratic, configuration of spaces, equipment, sensors, controllers and other components within a particular building. Descriptions of building configurations — called *metadata* — provide incomplete pictures of the information an application needs and are often inconsistent even within a single building. Existing standards for building metadata are ill-suited for reducing the implementation cost of energy efficiency applications because they are insufficiently expressive to capture the complex relationships between building components.

This chapter provides an overview of common building subsystems, motivates the need for "portable" applications that require minimal reconfiguration between buildings, and identifies the features of a schema for building metadata that enable portable applications. The chapter evaluates several state-of-the-art building metadata standards against these features and presents the case for a new metadata schema for buildings.

Building Systems Background

Most modern large commercial buildings contain a family of systems responsible for the building's operation, including the management of thermal conditioning, air quality, appropriate lighting, water, fire safety, security and use of space. As of 2012, roughly 14% of commercial buildings in the United States have building management systems (BMS) [108], which provide digital, programmatically accessible interfaces to *subsystems* that manage these aspects of building operation. Building subsystems are collections of equipment and infrastructure with monitoring and control capabilities. Buildings contain several subsystems; of these, heating, ventilation and air condition (HVAC) subsystems receive the most attention in this thesis due to their ubiquity and potential for savings (an average of 30% reduction in energy consumption [34]).

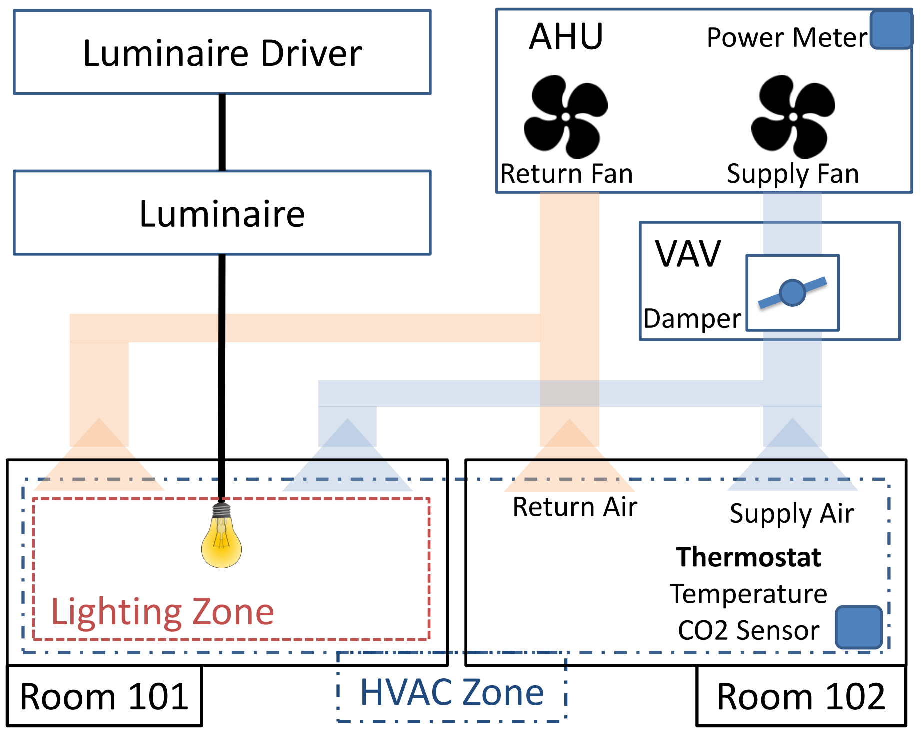


Figure 2.1: A simple example building that highlights the components to be modeled in a building schema.

An HVAC system is responsible for the transport and conditioning of air for the purposes of thermal comfort. Most HVAC systems have an air handling unit (AHU), which receives a mix of air from both the outside and inside of the building, conditions the air temperature using heating and cooling coils according to a setpoint, adjusts the air flow with a variable frequency drive (VFD) fan, and distributes the air along duct work to a series of terminal units. Heating and cooling coils are connected to external hot water and cold water systems that have their own components: chillers, condensers, boilers, valves, pumps and the associated set of sensors, setpoints, status and commands for monitoring and controlling them.

Terminal units are pieces of equipment that discharge air from an AHU into a set of usually adjacent physical spaces (e.g. rooms) called an *HVAC Zone*. Terminal units adjust one or both of air temperature and flow for the requirements of each HVAC zone. A common class of terminal unit is a variable air volume (VAV) box which can adjust air flow with the use of a controllable damper. Zone air is circulated back to the AHU along a return duct.

Figure 2.1 illustrates one such system. Rooms 101 and 102 are in the same HVAC zone, which is conditioned by a single VAV.

Although HVAC systems with AHUs and VAVs are common, the generic system described here is far from the only configuration. Buildings may have terminal units with additional functionality such as heating coils, or not have any terminal units at all. Other buildings may use radiant systems that use temperature-controlled surfaces instead of mechanically-driven air flow to control temperature.

Other building subsystems include:

- **Lighting Subsystems**: Modern lighting systems handle scheduling and occupancy-driven control in addition to automatically dimming or switching lights off for ambient lighting conditions or daylight harvesting. Controllers interact with LEDs and dimmable ballasts and monitor luminance and occupancy sensors through lighting control protocols such as DALI.
- **Electrical Subsystems**: large commercial buildings contain transformers and other substation equipment for regulating and monitoring the power distribution system. Electrical meters provide insight into the operation of circuits and subcircuits spanning floors and rooms all the way down to individual wall receptacles.
- **Spatial "Subsystems"**: while it is rare for spatial elements of a building to have a digital representation in a BMS outside of polygons on a rendered screen, they nonetheless intersect deeply with the operation of other building subsystems. Understanding how the spatial elements of a building relate to other building subsystems is useful for understanding their interactions; for example, the physics of a building's geometry affects the operation of HVAC systems and lighting.

The structure and function of building subsystems vary from building to building, as well as the APIs for reading from and writing to that equipment, the unique configuration of those subsystems and how they intersect with the design and construction of the building. Within a building, subsystems are often developed by different vendors, resulting in "siloed" systems that do not interoperate or share a naming convention for their components. As a result, developing energy efficiency applications requires developing an individual understanding of each building that is the target of the application. This is largely a manual, expert-driven process.

2.1 Case for Portable Applications

Energy efficiency applications need to have wide adoption in order to have a significant impact on societal energy usage. One limiting factor is the high degree of manual effort currently required to "port“ an application to a building. *Porting* an application means configuring an application to work on different buildings than those it was developed against; a *portable application* is one that requires little or no re-configuration when deployed on multiple buildings. Building applications vary in complexity from simple algorithms that only require a few directly-related data streams to algorithms using complex white- or grey-box models requiring detailed information about the construction and operation of a building.

This information typically must be acquired from diverse sources such as from a building automation system, architectural and mechanical drawings, operations and maintenance documents, and human input from the building operator. This information influences the logic of the application from building to building.

Because of the high degree of site-specific application logic and configuration needed to port an application, energy efficient strategies from the literature have not experienced any adoption, and the small number of applications actually implemented are done as "one-offs" – bespoke and proprietary implementations performed by a controls vendor for a particular building. This pattern produces implementations that suffer from bias and have little generalizability [54]. A recent evaluation of U.S. building energy benchmarking and transparency programs [69] found that "indications of [energy] savings should be considered preliminary... because of the limited period of analyses and inconsistencies among analysis methods for the various studies."

Portable applications present an opportunity to increase the adoption and deployment of energy efficiency strategies and analyses. To be portable, an application implementation needs to understand how to adapt its operation to the physical, logical and functional structures in a building. For example, a simple thermal model may want to understand the availability of data regarding temperature sensors, damper position, heating and cooling coil positions and air flow sensors, the locations of these sensors (e.g. which room and/or floor of the building) and the adjacency and exposure of those locations. The digital representation of these "things" (e.g. sensors, equipment, locations) and "relationships" (e.g. adjacency, influence, composition, measurement, process flow) is termed *metadata.*

2.2 Metadata Design Goals

The first question that a metadata system needs to answer is what it is trying to describe. This thesis was motivated by the failure of existing building metadata schemata to adequately name the elements of building subsystems and capture the relationships between those elements as required for a family of building analytics and control applications [17]. An effective metadata schema for buildings needs to succeed along three dimensions:

- Completeness: Can the schema name all of the entities and metadata information contained in building's BMS, as well as those expressed in canonical energy-, operations- and management-related applications?
- Expressiveness: Can the schema capture all important relationships that are contained in a building's BMS, and those expressed in canonical energy-, operations- and mangement-related applications?
- Usability: Can the schema unambiguously represent this information in a way that is easy to use for both the domain expert and the application developer? Can the schema

support automation with readable data formats and query tools? Can the schema be gracefully extended with new concepts?

Brick, developed in [9], is an open-source ontology providing a unified semantic representation of building assets, subsystems, and the relationships between them. The Brick ontology is the set of terms and rules used for constructing a Brick model, which represents the entities and relationships in a single building. A Brick model of a building is a labeled, directed graph in which the nodes are Brick entities and the edges are Brick relationships.

The Brick ontology has two components: the first is an extensible *class taxonomy* representing the recognized types of physical and logical entities in building. Examples of *physical* entities are pieces of equipment such as air handling units, variable air volume boxes, luminaires and sensing equipment as well as spatial elements such as rooms, floors and atriums. *Logical* entities do not necessarily have a specific physical representation. Examples include HVAC zones – a set of rooms all conditioned by the same terminal equipment – and data streams of sensors. The Brick class taxonomy is designed to gracefully extend to cover new equipment, points and concepts.

The second component of Brick is a minimal set of *relationships* that define the ways in which entities can be related to one another. Brick's relationships describe *structure* – how sets of things are assembled and located – and *process* – how sets of things compose to produce some result. Examples of structures described by Brick are spatial relationships (e.g. floors, rooms, staircases) and mechanical compositions (e.g. a VAV box has a damper). Examples of processes described by Brick are sequences of HVAC equipment (e.g. AHU feeding air to a VAV, feeding air to an HVAC zone) and control relationships (e.g. which thermostat's setpoint is used to condition the air in a given room).

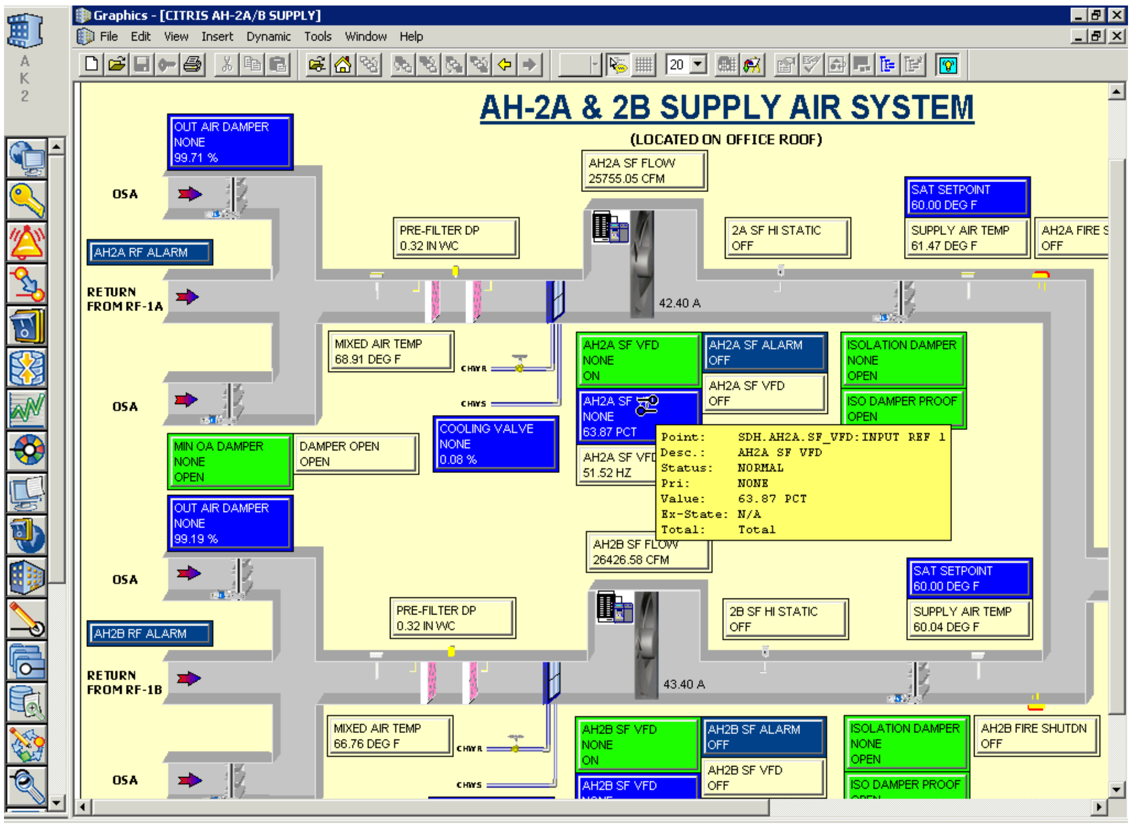
Brick does not aim be a comprehensive schema for the universe of arbitrary internet-of-things applications; instead, its design is guided by the sensors, equipment, entities, attributes and relationships that have been shown to be useful to applications from the published literature.

2.3 Existing Metadata Standards

Metadata for a building has several forms ranging from non-structured human-readable annotations to extensive and complex standards. A given building may not have any digital metadata, or it may have any number of differing digital representations of varying completeness and validity.

Metadata from Building Management Systems

One source of metadata is the monitoring and actuation networks contained in large commercial buildings, accessed through BMS (building management systems) or through SCADA (Supervisory Control and Data Acquisition) systems. BMS systems manage operational



```
Point:     SDH.AH2A.SF_VFD:INPUT REF 1
Desc.:     AH2A SF VFD
Status:    NORMAL
Pri:       NONE
Value:     63.87 PCT
Ex-State:  N/A
Total:     Total
```

Figure 2.2: Screenshot of a recently deployed building management system visualizing the internal operaton of an air handling unit. Note the heavy use of abbreviations and site-specific terms such as `SDH.AH2A.SF_VFD:INPUT REF 1`.

aspects of buildings involving the monitoring and control of equipment across building subsystems such as lighting, power, water, fire safety, security and heating, ventilation and air conditioning (HVAC).

These systems present a digital interface for reading from and writing to their underlying monitoring and actuation *points* – named values that represent the state of sensors, setpoints and actuators [23]. BMS point names are called *tags* and typically take the form of constrained alphanumeric strings. The role of tags in a BMS varies between BMS vendors and between buildings. At the least, tags uniquely identify a point within the BMS software for reference in visualizations and control logic, but provide no description of the point's function. More structured tags have components identifying the location, building subsystem, equipment and type of a point, in addition to providing a unique name.

One example is the tag `SDH.AH2A.SF_VFD:INPUT REF 1` in Figure 2.2. This tag has several components delineated by the `.` and `:` characters:

- `SDH`: indicates the name of the site (in this case, Sutardja Dai Hall on the UC Berkeley campus)

- `AH2A`: indicates the name of the air handling unit that contains this point
- `SF_VFD`: indicates that the point is for a variable-frequency fan (VFD) that serves as the supply fan for the air handling unit
- `INPUT REF 1`: indicates that the point represents the input speed of the fan

While tag naming and structuring conventions exist [28, 87], they do not lend themselves to a robust, generalizable and semantic representation of the building that could be used by a portable application. This is for two reasons. Firstly, BMS point labels are insufficiently expressive: labels only represent aspects of the building's control system that can be digitally read from or written to, e.g. sensor values and equipment commands. This precludes formal representations of structure and process beyond what is directly measured or controlled. BMS point labels cannot directly express the relationships between equipment across building subsystems, or the spatial relationships in a building (e.g., what rooms are in the same HVAC zone).

Secondly, point labels are not consistent: labels do not adhere to a common or consistent structure because they are designed to be consumed by humans; what structure exists is typically a building- or vendor-specific scheme. Further, web-based visualizations of building subsystems offered by BMS software are often the only representations of the functional and mechanical compositions of building subsystems, leaving no programmatically accessible representation that could be leveraged by an application (Figure 2.2).

Even with programmatic access to labels, data, and other descriptive information, scaling analytics or intelligent control across commercial buildings remains challenging. This is likely to be the case as long as the basic steps in interpreting the metadata involve labor intensive efforts by trained professionals with deep knowledge of building operations and specifics of each building. The difficulty and importance of normalizing BMS point labels has been noted in the literature [22, 61, 23, 110, 11].

Project Haystack

Project Haystack [85] is a commonly-used open source tagging system for describing buildings, equipments and points. A Project Haystack model of a building describes the equipment and points (generically referred to as "entities") in the building as a set of tag-value documents (Figure 2.3). Project Haystack defines a vocabulary of over 200 tags that define facts or attributes about entities. Project Haystack improves upon semi-structured point labels typical of BMS in its ability to define equipment and buildings as well as points, and to provide a machine-readable structure for representing distinct tagged attributes of those entities.

An entity has two flavors of tags: marker tags and value tags. Sets of marker tags define the type of an entity and have no associated value. All entities have at least one marker tag that defines the base type of the entity: `site` for a single building, `equip` for physical or logical equipment in a site, and `point` for sensor, actuator and setpoint values for equipment.

```
# marker tags
equip
ahu
hvac
rooftop

# value tags
id: @2180b666-7032054c
dis: "RTU-1"
siteRef: @2180b666-430b2363
elecMeterRef: @2180b666-7032054d
```

Figure 2.3: A sample rooftop unit entity represented in Haystack. The entity is related to a site and an electric meter. Its display name is “RTU-1”. Example pulled from [84].

For example, a zone temperature sensor has the marker tags `zone`, `temp` and `sensor`. Further marker tags on an entity refine its type: all air handling units have the `ahu` tag, and the addition of the `rooftop` tag affirms that the equipment is a rooftop unit, a smaller and more compact kind of air handling unit. This is similar to the notion of “subclassing” from object-oriented type systems, but lacks a formal structure. Figure 2.3 contains a Project Haystack representation of a rooftop unit.

Value tags define attributes of entities – such as identifiers, timezone, engineering units and geo-coordinates – and their relationships to other entities. Relationships between entities are identified by a special kind of value tag called a *ref tag*; these can be identified by their `-Ref` suffix. The entity possessing the ref tag is the subject of the relationship, the name of the ref tag indicates the nature of the relationship, and the value of the ref tag is the identifier of the entity who is the object of the relationship. There are a limited number of ref tags supported by Project Haystack: `equipRef` associates a point with a piece of equipment, `ahuRef` associates an air handling unit with a VAV or chiller, `elecMeterRef` associates an electric meter with a piece of equipment, and `siteRef` associates an entity with a site.

The current set of tags lacks or does not fully describe key aspects of buildings such as spatial elements, lighting equipment and electrical subsystems [17]. While deficiency of tags can be addressed by future updates, a more fundamental challenge with a tag-based scheme is it is difficult to capture sufficient context to disambiguate the meaning of certain sets of tags. There is a tension between how generally a tag can be used and how well defined it is. Consider an entity with the tags `gas` and `heat`: without additional context, it is unclear if the entity uses gas as a fuel to heat another substance, or if the entity heats up gas (such as for preparing LP-gas for consumption). It can also be ambiguous when to use certain tags to describe an entity without some external set of rules: when tagging a room air temperature sensor, should the `zone` tag be included in the absence of a `room` tag? There is no mechanism to enforce the “correct” grouping of tags to annotate entities in a Project Haystack model; as a result, many Project Haystack models use proprietary or site-specific tags to resolve local ambiguities. The tagging system gives users this flexibility at the cost of such models lacking a common, site-agnostic structure against which portable applications can be written.

Furthermore, Haystack relationships lack expressive the power to generalize across subsystems. The ref tags defined by Project Haystack that can relate equipment can only express narrow, HVAC-specific associations:

- `ahuRef`: associates VAVs with their upstream air handling units
- `chilledWaterPlantRef`: associate an entity with a chilled water plant
- `hotWaterPlantRef`: associate an entity with a hot water plant
- `steamPlantRef`: associate an entity with a steam plant
- `device1Ref`, `device2Ref`: associate entities with a network connection
- `elecMeterRef`: associate an entity with an electric meter
- `equipRef`: associate an entity (usually a point) with a piece of equipment

These relationships only capture that two entities are related, and do not capture the nature of that relationship. For example, a VAV may have an `ahuRef` tag with a value of its upstream AHU, and a supply air flow sensor may have a `equipRef` tag with the value of that air handler unit. From these tags, the model cannot express how the VAV and supply air flow sensor relate to one another in the flow of air through the system: does the supply air flow sensor occur before or after the cooling coil inside the AHU?

Project Haystack models can be accessed through a REST API and a filtering query language[1]. The query language provides basic mechanisms for identifying timeseries points using the associated tags, but does not provide a way of traversing `ref` tags for the purpose of exploring the structure of a Haystack model. This limits the expressiveness of queries against the model.

Haystack Tagging Ontology (HTO) [27] maps the Haystack tags to an ontology, with each tag corresponding to an ontology class. Thus, HTO is able to combine the flexibility of tags and the formal modeling of ontologies to define essential BMS metadata and the relationships between entities. However, HTO confines the ontology to the defined tags, and the building entities which are a collection of tags (e.g. zone temperature sensor) are not modeled. HTO also does not provide a way to compose complex subsystems in a building and relies on Haystack tagging for mapping raw metadata to the ontology.

Correct interpretation of a Haystack model requires an out-of-band understanding of how a tag or set of tags is meant to be interpreted. The informality of the Haystack model inhibits what the model can express consistently, which limits the portability of applications implemented against a Haystack model.

[1]`http://project-haystack.org/doc/Ops`

Industrial Foundation Classes

IFC [12] is a standardized Building Information Model (BIM) that developed from the need to have a common exchange model for 3D architectural drawings needed for a building's construction. IFC models capture structural information, including space-related information such as floors, rooms and zones, and exhaustively describes the mechanical composition of building subsystems. For an HVAC system, IFC describes not just AHUs and VAVs, but also ducts, flanges and other mechanical components not directly measurable or controllable.

In contrast to BMS point labels and Project Haystack tags, the metadata structure captured by IFC is not intended for the operation of a building, and thus lacks much of the required vocabulary. Recent versions of the IFC standard include references to generic sensor types (`IfcSensorTypeEnum`) which can be associated with the spaces the sensor covers. However, the IFC standard does not include explicit mechanisms for describing the functional role of sensors, such as whether a temperature sensor measures supply, return or exhaust air. There is also no common way of adding new vocabularies compliant to existing ones. The IFX 2x2 schema also contains descriptors for building controllers[2], which describe at a high level the existence of alarms, events and schedules.

Ultimately, the IFC standard is not intended for authoring portable applications. The IFC standard does not define enough of the components involved in the operational aspects of a building, nor the relationships between those components. Section 5 explores using the information in an IFC model to create a partial Brick mdoel.

Semantic Web and Ontologies

Semantic Web is a framework promoting common data formats and exchange protocols that facilitate the sharing and reusing of data across systems and domains. The Semantic Web describes data using controlled vocabularies called ontologies: an *ontology* is the set of formal names, concepts, definitions and relationships that constitute a domain of knowledge; in other words "an explicit specification of a conceptualization [45]". Originally intended for annotating the Web documents [15], the semantic web has since expanded to include ontologies pertaining to biology [8], IoT [46], and energy management [115] to control the complexity of the domain information.

Ontologies are essentially a structure for defining structured data[3]. This gives ontologies a degree of expressiveness not possible in other metadata schemata such as Project Haystack. This thesis in part investigates the effectiveness of ontologies in capturing the complexity and heterogeneity of building metadata.

A number of ontologies have been proposed for smart homes and buildings. Most of these ontologies focus on realizing specific applications like controlling things [21], energy management [57], or automated design and operation [79]. Daniele et al. [31] combined

[2]`http://www.buildingsmart-tech.org/ifc/IFC2x4/rc2/html/schema/ifcbuildingcontrolsdomain/content.htm`

[3]meta-metadata

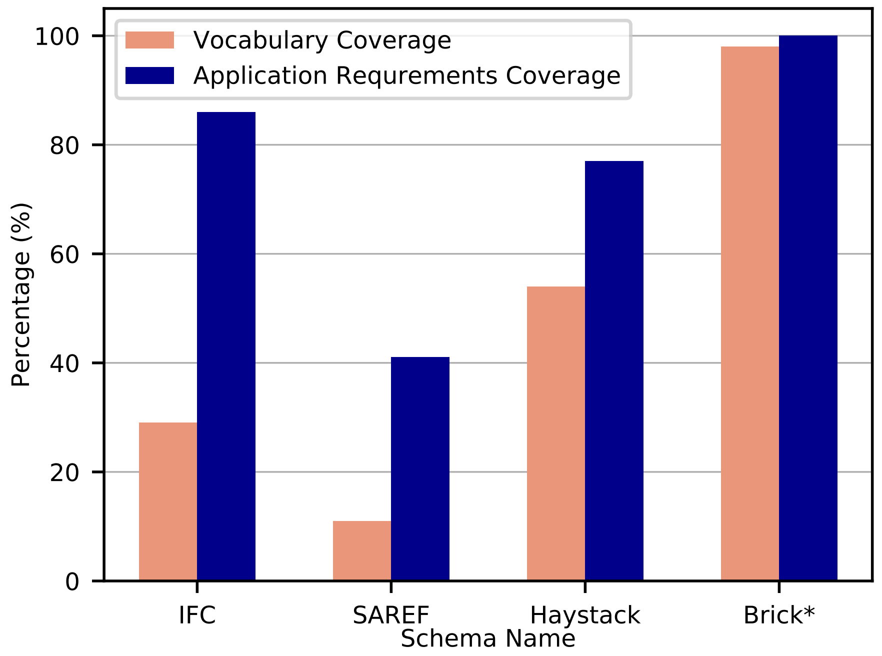


Figure 2.4: Comparison of different schemata for buildings from [17]. The paper used 89 applications (apps) and three buildings to evaluate IFC, SAREF and Haystack. The results for Brick are described in Chapter 6.

these ontology modeling efforts in collaboration with industry to create a simple but unified model called Smart Appliances REFerence (SAREF). They identify 20 recurring concepts in homes and buildings across these ontologies, and lay out the steps to convert SAREF to a custom ontology. These common concepts, however, do not effectively cover the diversity of devices and equipment in buildings [17] because their goal was to capture generic sensor and smart devices rather than building operations where domain-specific information is required. Ontology representations of IFC [13] and Haystack [27] also exist, but only model the limited semantic structure of their source metadata schemes.

The BOnSAI [97] smart building ontology describes the functionality of sensors, actuators and appliances as well as how they interact and effect their physical environment. However, it fails to capture the interactions and relationships between the sensors and other building assets. Hence, it lacks a system-level view of the building infrastructure necessary for many applications [17]. Further, the vocabulary does not describe the mechanical or functional compositions of critical building subsystems like HVAC and lighting.

Analysis of Existing Schemata

Bhattacharya et al. [17] performed a comparison of Haystack [85], IFC [12] and SAREF [31]. The paper uses 89 building applications from eight representative applications categories published in the literature as a baseline to compare different schemata. The results show that relationships between different pieces of information are essential to enable interoperability and portability of building applications over three buildings.

They use three metrics to measure the effectiveness of each schema: (i) the ability to completely map BMS metadata from three existing buildings to the schema, (ii) ability of the schema to capture the relationships required by applications, and (iii) the flexibility of the schema to deal with uncertainty as well as their extensibility to new concepts. Figure 2.4 presents the comparison across Haystack, IFC, SAREF and Brick for metrics (i) and (ii).

Among the three existing schemata, Haystack shows the best vocabulary coverage as it is a tag-based model where tags can be arbitrarily combined. IFC is the most complete in describing application relationships as its model captures the building subsystems and the dependencies between them. SAREF scored the lowest for both metrics because it models the common concepts across different models and systems instead of comprehensively modeling buildings. None of the three schemata succeed on all metrics.

2.4 Summary

This chapter has outlined the need for portable applications in increasing the penetration of energy efficiency-oriented building applications. Portable applications operate over digital representations – metadata models – of buildings that provide building applications a mechanism for querying models for the information they need to run. The digital representation must be *complete* (can it name all of the entities required by applications), *expressive* (can it capture all of the relationships required by applications) and *usable* (can it be used to build real, portable building applications).

The following chapters discuss the design and implementation of Brick, a new ontology-based metadata schema that captures the entities and relationships necessary for effective representations of buildings and their subsystems. Brick describes buildings in a machine readable format to enable programmatic exploration of different operational, structural and functional facets of a building. Hence, the diverse set of BMS information can be represented using Brick, and applications developed based on the Brick schema can be directly deployed on those buildings in a portable manner. This thesis explores the design of a relationship-focused, graph-based building metadata schema – Brick – that facilitates the implementation of portable applications, thereby enabling the mass customization and implementation of energy efficiency analytics and applications.

Chapter 3

Design of the Brick Schema

Having motivated the need for a new, standard metadata schema that enables the development and deployment of portable applications, we delve into the design of the **Brick schema** which addresses these needs. We define Brick from the bottom up, starting with *Tags* and *TagSets* which are the fundamental named and defined elements in Brick. We then discuss how to assemble TagSets into a *class hierarchy* and define a family of *relationships* that can express the appropriate context for portable applications. Together, these form the initial release of Brick.

The contents of the Brick class taxonomy are informed by ground truth information from six different buildings from the US and Europe. Together, the six buildings consist of 17,700 data points and five different BMS vendors. To evaluate the *completeness* and *usability* of the Brick schema, we implement Brick models for each of the buildings and measure how well Brick captures the ground truth metadata.

3.1 Core Brick Concepts

We define the key abstract concepts involved in the design of Brick and provide examples.

Definition 3.1: Entity

An *Entity* is an abstraction of any physical, logical or virtual item; the actual "things" in a building.

Brick defines how entities can be classified and related to one another. There are several flavors of entities:

- Physical Entities: anything that has a physical presence in the world. Examples are mechanical equipment such as air handling units, variable air volume boxes, luminaires and lighting systems, networked devices like electric meters, thermostats, electric vehicle chargers and spatial elements like rooms and floors.

- Virtual Entities: anything whose representation is based in software. Examples are sensing and status points which allow software to read the current state of the world (such as the value of a temperature sensor, the speed of a fan or the energy consumption of a space heater), and actuation points which allow software to write values (such as temperature setpoints or brightness of a lighting fixture).
- Logical Entities: entities or collections of entities defined by a set of rules. Examples are HVAC zones and Lighting zones. Concepts such as class names, Tags, and Tagsets which help define Brick also fall into this category.

Brick provides rules for how to define, classify and describe entities. These rules are built upon *Tags* and *TagSets*, and take the form of an *ontology*.

Definition 3.2: Tag

A *Tag* is an atomic fact or attribute of an entity. Examples of tags are `sensor`, `setpoint`, `air`, `water`, `discharge`, `leaving` and `vav`.

We borrow the concept of Tags from Project Haystack [85] to preserve the flexibility and ease of use of annotating metadata, but do not rely on tags alone to determine the type of an entity. Sets of tags insufficiently represent the type of an entity because there is no mechanism to differentiate between different interpretations. To address this, Brick combines sets of tags into identifiers termed *TagSets*

Definition 3.3: TagSet

A *TagSet* is a named collection of Tags with a clear, unambiguous definition. They constitute the set of valid types used to classify entities in Brick; in this context, TagSets are referred to as *class names*. Examples of TagSets are `Variable Air Volume Box`, `Air Flow Sensor` and `Hot Water Discharge Temperature Sensor`.

In Brick, the type of an entity is determined by its class name; class names have formal definitions in order to resolve any ambiguity about their meaning or intended use. This leaves tags to be used for annotation and discovery rather than for definition. Brick defines classes `Boiler`, `Hot Water Coil` and `Hot Water Discharge Temperature Sensor` that all have the `hot` and `water` tags, but use hot water differently.

Brick implements Tags, TagSets and their definitions with an *ontology*.

Definition 3.4: Ontology

An *Ontology* is the set of formal names, concepts, definitions and relationships that constitute a domain of knowledge. Ontologies can express constraints on how concepts and relationships can be used.

Brick uses an ontology to:

- define the set of Tags and TagSets, and rules to how tags can be used. As an example, an ontology can prevent an entity from having both the `sensor` and `setpoint` tags by defining them as being mutually exclusive.
- provide definitions of classes (e.g. an `Air Handling Unit` is a class describing a "device used to regulate and circulate air as part of a heating, ventilating, and air-conditioning (HVAC) system")
- define the structure of class hierarchy and the nature of inheritance between classes (described in Section 3.2), for example the `Hot Water Discharge Temperature Sensor` class is a subclass of a more generic `Water Temperature Sensor` class which is a subclass of a more generic `Temperature Sensor`, and so on.
- define the set of *relationships* (described in Section 3.3)

Comparison with Tag-Based Metadata

Brick's design decision to determine the type of an entity with a well-defined TagSet rather than a set of Tags offers a distinct advantage over classical tag-based metadata systems such as Haystack. In tag-based metadata systems, ambiguous definitions for a set of tags must be resolved with additional tags. For example, Project Haystack defines the `hotWaterHeat` tag that can be applied to an entity that uses hot water to perform a heating action. Per Project Haystack's documention, the `hot` and `water` tags are only for describing points that "indicate control or measurement of hot water"[1]. The `hotWaterHeat` tag exists because the existence of the `hot` and `water` tags on an entity is not enough to determine the function of that entity. This impacts the discoverability of entities: intuitively, a user might look for all entities related to a building's hot water system by searching for entities with the `hot` and `water` tags, but this will not find entities with the `hotWaterHeat` tag. In Brick, all entities related to the hot water system can have the `hot` and `water` tags while having distinct and well-defined types determined by class names.

[1]`https://web.archive.org/web/20181211044618/https://project-haystack.org/tag/hot`

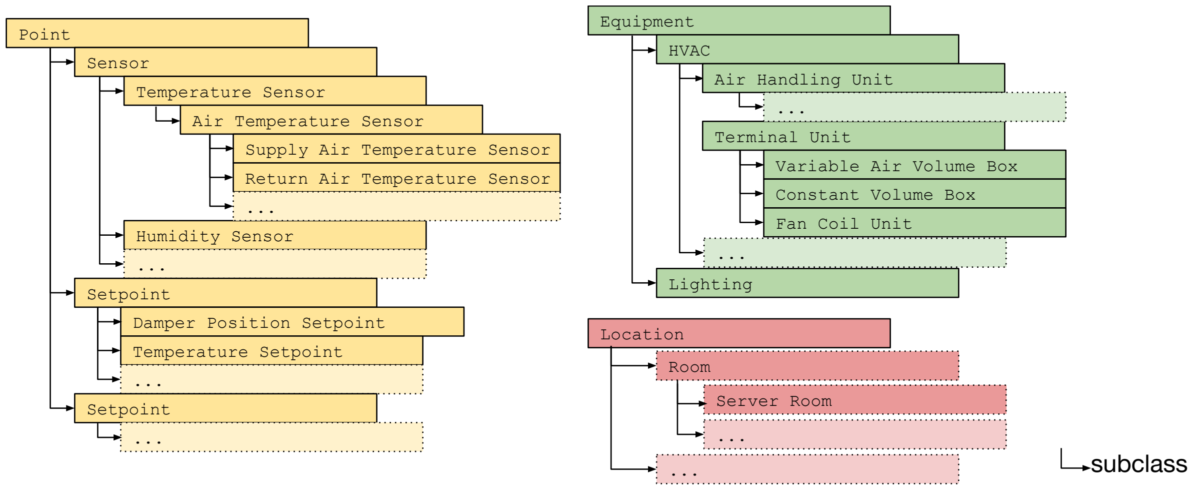


Figure 3.1: A subset of the Brick class hierarchy. All Brick classes are subclasses of least one of the base classes, which have no super classes: `Location`, `Point` and `Equipment`. The fourth base class `Resource` is not pictured.

3.2 Class Hierarchy

Several high level concepts provide the scaffolding for Brick's class hierarchy. There are four root classes from which all Brick classes are derived:

- `Point` is the root class for the hierarchy representing points of telemetry (representing a single timeseries) and actuation. It has five immediate subclasses that define the further roles of points. These definitions are intended as broad guidelines
  - `Sensor`: outputs of transducers that record the state of the physical world; typically read from; example `Air Temperature Sensor`
  - `Setpoint`: control points used as a target value in a feedback-driven control system; typically written to; example `Air Temperature Heating Setpoint`
  - `Command`: control point that directly affects the state of equipment; typically written to; example `Cooling Valve Command`
  - `Status`: relating to the current logical status of equipment; sensors that are not transducers; typically read from; example `Damper Position`
  - `Alarm`: high-priority indicators; typically read from; example `Water Loss Alarm`

  The `Point` class can refer to virtual sensors as well.

- `Equipment` is the root class for the hierarchy representing mechanical equipment across all types of building subsystems: HVAC, elighting, electrical, fire, security, etc. Examples include air handling units, variable air volume boxes, luminaires, light switches, breaker boxes, meters, fire alarms and security cameras. `Equipment` are controlled by their points.

- `Resource` is the root class for the hierarchy representing the resources and substances that need to be referred to. *Resources* and *Substances* are the physical properties or substances that are the subject of monitoring and control points. Examples of resources are the position of a damper and the operating mode of a thermostat (e.g. heat-only, cool-only, off). Examples of substances are water, air and electricty and their subclasses. This hierarchy defines the `Hot Water` class that is the substance created by a `Boiler` class from the `Equipment` hierarchy. This hierarchy also defines logical substances such as `Fan Speed`.

- `Location` is the root class for the hierarchy representing the spatial domain of a building. This includes physical elements such as rooms, floors and buildings as well as logical elements such as HVAC and lighting zones.

Each of these root classes has a multi-level hierarchy of subclasses attached to it. Subclasses extend the definitions of superclasses with additional properties and specifications. For example, consider the `Point` class hierarchy in Figure 3.1: the `Temperature Sensor` class extends the `Sensor` class with the additional property that the sensor measures temperature. The specificity of a class increases further down the hierarchy: the `Air Temperature Sensor` class defines `Air` as the substance whose temperature is being measured, and the `Return Air Temperature Sensor` and `Supply Air Temperature Sensor` classes further refine the definition with the position of the sensor within the HVAC process.

The design of the Brick class hierarchy allows it to be expanded in future releases to include new classes of equipment, locations, or points. The set of root classes can also be expanded to cover other domains in the built environment, such as networking or administration.

The use of an ontology to define the hierarchy lets Brick define rules to resolve some ambiguities. For example, it is common in a domain to use multiple terminologies for the same entity. For example, in HVAC systems, `Supply_Air_Temperature` and `Discharge_Air_Temperature` are sometimes used interchangeably. We identify these synonyms from our ground truth buildings, and mark the corresponding tagsets as being equivalent classes in Brick using the ontology. Synonyms ease translation from raw building metadata and allow for cultural differences in terminology without affecting application functionality. Note that the class hierarchy is not restricted to a tree structure, and can use multiple inheritance when appropriate. For example, a desk lamp class can be a subclass of both the lighting system and office appliance classes.

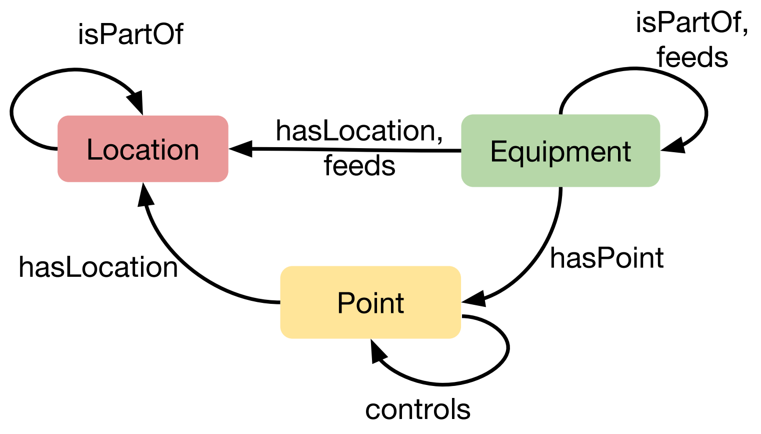


Figure 3.2: Information concepts in Brick and their relationship to a data point.

3.3 Relationships

Relationships express how Tags, TagSets and entities interact and are associated with each other. Brick defines a finite set of relationships that can express the relationships required by applications, and do so in a portable manner (Table 3.1).

Definition 3.5: Relationship

A *Relationship* defines the nature of a link between two related entities. Examples of relationships are encapsulation (one entity is contained within another), sequence (one entity takes effect before another in some process) and instantiation (one entity's type is given by another entity).

Definition 3.6: portability

Portability is the quality of a relationship that makes it generalizable to different settings.

Well-defined, portable relationships are key to reducing the risk of inconsistency across buildings. To this end, Brick's relationships express the following dimensions:

- **Composition:** Composition is relevant for equipment (relationships that define what equipment an entity is a part of, or what equipment is a part of it) and locations

(relationships that capture relative and absolute position of entities; for example, which building, floor and room an entity is in, but also where in the room it is)

- **Influence:** relationships that define what points affect the behavior of other points and equipment, and relationships that define what equipment an entity is connected to, and how it they are connected; for example, how AHUs feed VAVs, how hot water or cold water systems integrate with terminal units, which meters are connected to which equipment
- **Monitoring:** relationships that define what measures the entity or what it measures
- **Taxonomy:** relationships that define the class hierarchy, tags, tagsets, and how to create "instances" of Brick classes that relate to entities in a Brick model.

Table 3.1 lists the core relationships defined in the Brick ontology. Figure 3.2 shows some of these relationships and how they relate to three Brick base classes – `Equipment`, `Point` and `Location`. The ontology is responsible for enforcing the types of the *subject* and the *object* for each relationship. The subject of a relationship is the entity that possess the relationship's indicated property, and the object of a relationship is the entity that the relationship is acting upon. For usability, Brick uses the ontology to define a corresponding inverse relationship for some relationship so that users can express relationships in any direction they prefer.

Brick provides these constraints as a set of guidelines for Brick model developers to aid in keeping Brick usage consistent between building models. For example, the object of `hasPoint` must be an instance of a class in the `Point` hierarchy, and the subject of `isLocationOf` must be an instance of a class in the `Location` hierarchy Therefore, a VAV (a subclass of `Equipment`) can have `Points` like `Zone Temperature Sensor, Discharge Air Flow Setpoint, Reheat Valve Command`, and it can have other equipment as subcomponents such as `Damper` and `Reheat Valve`.

The set of Brick relationships can generalize across building subsystems: An example of a portable relationship is `feeds`. The `feeds` relationship captures the different *flows* between entities such as equipment or locations in the building, such as the flow of air from AHU to VAV, the flow of water from a tank to a tap, or the flow of electricity from a circuit panel to an outlet. In contrast, Project Haystack's `ahuRef` relationship explicitly connects equipment and points to an air handler unit. This lacks portability because it is only valid in the context of AHU-based HVAC systems.

Defining Application Context With Relationships

When an application executes on a building, it pulls the necessary *context* from the Brick model describing that building. *Context* is the set of entities related to the application, which have been identified both by their type and by their relationships.

Consider a fault detection application that operates on variable air volume boxes (VAVs) running on the sample building in Figure 3.3. This application needs to know the equipment

Dimension	**Relationship (Inverse)**	**Interpretation**	**Endpoints Subject / Object**
Composition	hasPart (isPartOf)	`A` has some component or part `B` (typically mechanical)	Equip. / Point Equip. / Equip. Location / Location
	isLocationOf (hasLocation)	`A` physically encapsulates `B`	Location / Point Location / Equip.
Influence	controls (isControlledBy)	`A` determines or affects the internal state of `B`	Point / Point
	feeds (isFedBy)	`A` "flows" or is connected to `B`	Equip. / Location Equip. / Equip.
	hasInput (isInputOf)	controller `A` has input `B`	Controller / Point
	hasOutput (isOutputOf)	controller `A` has output `B`	Controller / Point
Monitoring	hasPoint (isPointOf)	`A` is measured by or is otherwise represented by point `B`	Equip. / Point Location / Point Resource / Point
Taxonomy	type	`A` has type `B`	Equip / Class Point / Class Location / Class
	subClassOf (isSubClassOf)	class `A` is a subclass of class `B`	Class / Class
	hasTag (isTagOf)	class `A` has tag `B`	Class / Tag

Table 3.1: List of the Brick relationships and their definitions. All definitions follow the form `A <relationship> B`, where `relationship` is the first one listed, not the inverse. All Brick relationships are asymmetric, and most are transitive. If a relationship $\rightarrow$ is transitive, then if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$ is a valid relation. Asymmetric simply means that if $A \rightarrow B$, then $B \rightarrow A$ is invalid.

in and around the VAV (such as variable frequency drive fans and dampers to control air flow), and the points that describe the state of that equipment and sense the temperature and flow of the air moving through the VAV. Further, the application needs information about the air handling unit (AHU) that is upstream of the VAV, as well as the set of rooms that are served by the VAV.

The context here includes how equipment is monitored and controlled, how the equipment influences air, what spatial elements of the building are immediately affected and how equipment is positioned in an HVAC process. Applications define context by using *queries* against a Brick model; this mechanism is covered in depth in Chapter 4. The next chapter discusses the requirements of a query language for a Brick model, and how this is implemented for Brick.

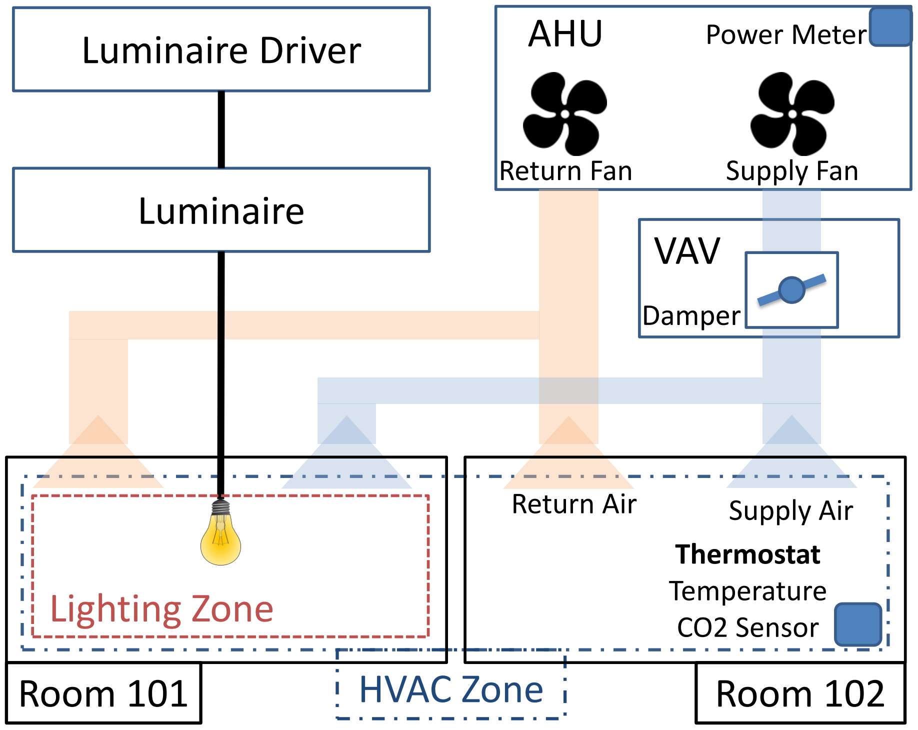


Figure 3.3: A sample building with simple HVAC and lighting subsystems, shown with their associations to physical spaces in the building.

Directed Labeled Graph Structure

Brick's concepts of entities and relationships can be interpreted as nodes and edges in a directed, labeled graph. A relationships' subject and object are the source node and destination node for a directed edge; the label of that edge is the name of the relationship. Nodes are also named.

This abstraction describes Brick models – the set of entities and relationships that make up an instance of a building – as well as the Brick ontology itself. Figure 3.4 shows the Brick model – entities and relationships – for the HVAC and lighting system processes for the sample building in Figure 3.3. While this example only shows the building-related entities, the graph abstraction extends to how Brick is defined: the definitions of Tags and TagSets, the relationships of Tags to TagSets, the definition of relationships and of constraints on the use of relationships can all be represented as a directed, labeled node-edge graph. How this is achived is explored in Chapter 4.

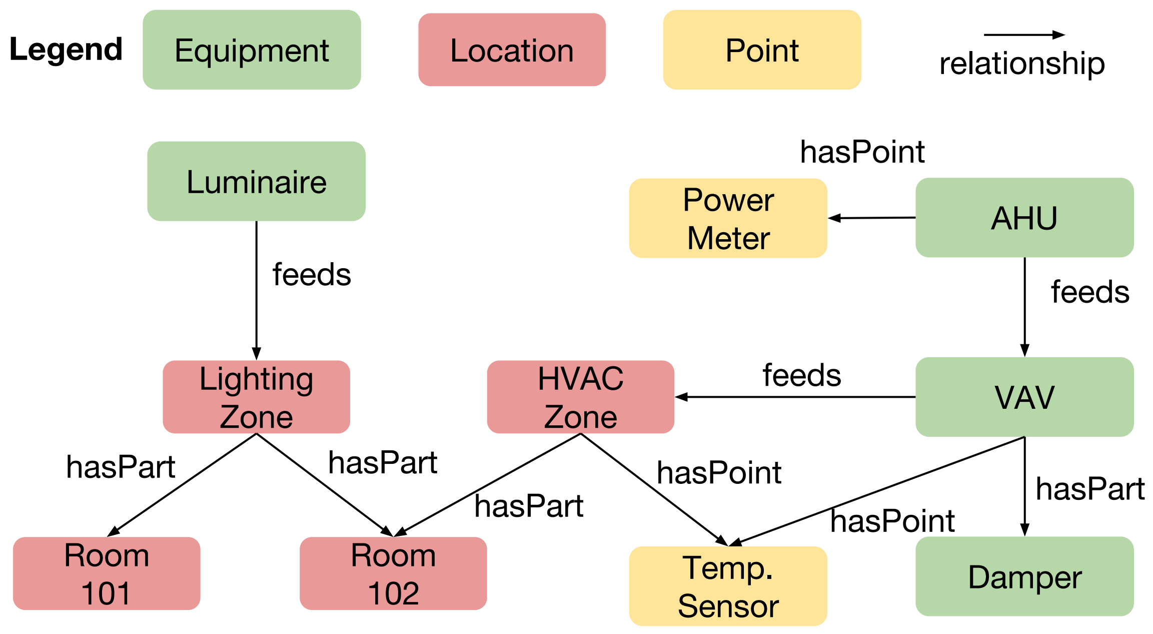


Figure 3.4: Brick classes and relationships (constituting a Brick model) for the HVAC and lighting processes in the sample building in Figure 3.3.

3.4 Evaluation of Relationship Coverage

To evaluate how well the set of Brick relationships covered the context required by real applications, we implemented a representative application from each of the eight canonical application categories identified by Bhattacharya et al. [17]. Table 3.2 shows the set of entities and relationships that each application needed to refer to. Brick's relationships are sufficiently expressive to capture all application requirements.

We present an overview of each of the eight canonical application categories and provide a brief description of each of the applications implemented in Table 3.2[2]

- **Occupancy Modeling**, typified by [52]: Occupancy modeling applications use occupancy data to drive analytics or control processes. A challenge for occupancy-driven applications is aggregating occupancy information from disparate building subsystems, such as the triggers for occupancy-driven lights, the schedule defined on a thermostat, or independent wireless sensing systems. The application described in [52] uses occupancy sensors in conjunction with device-specific energy meters to develop a model for predicting energy usage and building occupancy. The queries for this application are in Figure A.4.

[2]table located at end of chapter

- **Energy Apportionment**, typified by [51]: Energy apportionment applications use multi-modal sensor data to attribute energy consumption patterns with individuals. A challenge for energy apportionment applications is understanding the relationship between device and equipment state and the metering infrasturcture; for example, if a VAV's heating coil is 60% open, which meter should an application read to understand the energy consumption? The application also needs to be able to connect device and equipment activity with occupancy sensors. The application described in [51] tracks user activity with an ubiquitous sensor-based system that can correlate user interactions with heating and lighting systems with energy consumption. The queries for this application are in Figure A.1.

- **Web Displays**, typified by [10]: Web displays are a class of application that presents a user-facing dashboard that aggregates building data to provide insight; a classic example are BMS web interfaces (Figure 2.2). These applications need to be able to describe sequences of equipment and monitoring and controlling points for multiple building subsystems, as well as how those subsystems interact and intersect (e.g. what are the lighting resources and HVAC resources for a given room?). The [10] application implements a virtual thermostat in software that provides users more visibility into their energy consumption and more control over thermal comfort. The queries for this application are in Figure A.6.

- **Model Predictive Control (MPC)**, typified by [99]: MPC is an advanced process control methodology that uses dynamic models of physical processes to predict future states and to determine immediate control actions. A challenge for MPC applications is identifying the points needed to create data-driven models and take action on the output of the MPC process. The application described in [99] builds a simple linear thermal model for each zone in a building, taking into account solar gains and heating and cooling system actions. The queries for this application are in Figure A.2.

- **Participatory Feedback**, typified by [60]: Participatory feedback applications incorporate user preference into control decisions that affect the user's environment. The application described in [60] implements persionalized lighting controls. To do this, it needs to be able to relate the control points of the lighting system to the spatial area that is illuminated. The queries for this application are in Figure A.7.

- **Fault Detection and Diagnosis (FDD)**, typified by [90]: Fault detection and diagnosis applications monitor equipment through their points, build a model of nominal operation, and notify the user when the system exhibits abnormal behavior. Fault diagnosis involves developing a causal explanation for the fault. A challenge for these kinds of applications is discovering the points that describe equipment behavior as well as "downstream" effects; this involves identifying the set of equipment, points and other entities that could be affected by an equipment fault. The application described in [90] defines a rule-based method for performing fault detection in air handler units. This application requires a description of the HVAC subsystem that includes the sequences of equipment, which spaces are affected by which HVAC equipment, and the points exposed by the BMS that can provide insight into the HVAC process. The queries for this application are in Figure A.8.

- **Non-Intrusive Load Monitoring (NILM)**, typified by [66]: NILM applications use electrical meters (such as full building meters) and equipment and device state indicators to deaggregate the energy consumption of equipment or identify collections of equipment from an aggregate meter. This is a form of *system identification*. Challenges facing NILM applications include describing electrical submetering infrastructure and device and equipment state. The application described in [66] uses a device-level control system and circuit-level power measurements to perform NILM in a residential setting. The queries for this application are in Figure A.5.

- **Demand Response (DR)**, typified by [109]: Demand Response applications adapt the control regime of a building in response to changes in the price of electricity (for an overview, see [4]). Simple DR applications react by load shedding through widening setpoint deadbands and dimming lighting fixtures; advanced DR applications create models of building subsystem behavior to understand how to balance other concerns, such as the thermal comfort of occupants with the price of cooling the building on a hot day. The application described in [109] coordinates control of diverse plug loads during a demand response event when the price of energy has risen dramatically. A challenge for this application is describing the types of of device and equipment plug loads so that the demand response controller can reason about which loads to shed. The queries for this application are in Figure A.3.

Table 3.2 illustrates that the entities and relationships required by each of the chosen applications can be completely defined by the Brick class hierarchies (`Point`, `Equipment` and `Location`) and Brick relationships. Chapter 6 evaluates Brick's completeness and expressiveness by implementing these applications against a set of real buildings.

3.5 Control Sequences

A control sequence is the logic determining the behavior of equipment. Most representations of buildings do not capture effective descriptions of control sequences; what is found are either mathematical descriptions of implemented control algorithms, or select variables within the logic implementation that are exposed as points in building systems. Some points' values are measurements of physical properties, some are results of calculations and others are configuration parameters used to control physical devices. The flow of these control signals is key for understanding buildings operations. Users rely on the flow of control to interpret values (e.g. does the value of the `Zone Temperature Sensor` make sense given the value of the `Heating Temperature Setpoint`, `Cooling Temperature Setpoint` and `Air Flow`

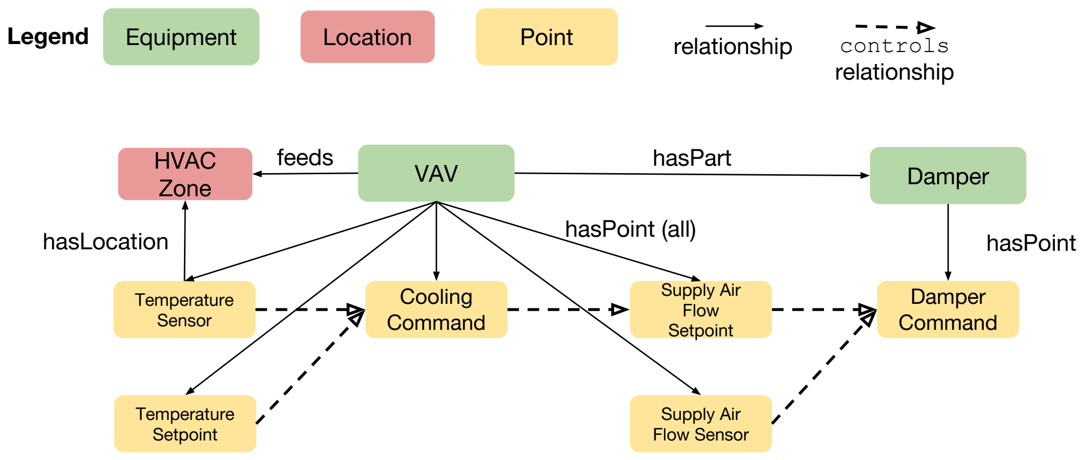


Figure 3.5: Control flow example of a simplified VAV. A VAV has points related to equipment control to adjust its feeding zone's temperature. A point's value is often determined by other points' values. Such dependencies are modeled as `controls`.

`Setpoint` involved in the same control loop?) and to determine how controls affect the building (e.g. which BMS points are responsible for affecting the temperature and airflow in a given room?).

Representations of control logic vary wildly. In many older buildings, control logic is embedded in physical controllers distributed throughout a building. Some vendors will provide visualization tools to represent their proprietary control logic, and others use proprietary programming languages. These are often accompanied with documents providing human-readable specifications of the control logic's behavior.

Some compelling representations of control logic include Simulink Simscape and Modelica. Simulink Simscape [92] provides multi-domain simulation of control logic defined with mathematical models, but is designed for simulations rather than for integrating with real physical systems. Modelica [40] is an object-oriented language and execution environment for modular simulations and has current development efforts focusing on building control and simulation [112]. These representations, however, are only used in simulation and not designed for BMS operation. MLE+ [14] and BCVTB [111] have created co-simulation environments where control simulation logic can be deployed in real buildings with BMS. They are designed for experimental evaluation of control algorithms and are not meant for production operation of buildings.

Brick does not currently model the control logic in building systems; rather, it describes the dependencies between sensors, actuators, commands, setpoints and related equipment and spaces. Instead, it is intended that control algorithms will be written over Brick, using Brick to describe the set of involved resources and relationships.

Brick models the control dependencies using the `controls` relationship between points. When a point's value is used for another point's value determination, we say that the former one `controls` the later one. Figure 3.5 is an example with a simplified version of VAV control. An AHU provides temperature-controlled air to VAVs, which control their associated zones' temperature by changing the amount of air flow. When the zone's temperature is lower than its corresponding setpoint, the VAV increases the supply air flow controlled by its damper. To be more specific, `Cooling_Command` increases proportionally to the difference between `Zone_Temperature_Sensor` and `Zone_Temperature_Setpoint`. `Cooling_Command` determines `Supply_Air_Flow_Setpoint` and the difference between `Supply_Air_Flow_Setpoint` and `Supply_Air_Flow_Sensor` determines the value for `Damper_Command`. `Damper_Command` affects its damper's state that controls actual air flow. We model these dependencies with `controls` such as "`Zone_Temperature_Setpoint controls Cooling_Command`" and "`Zone_Temperature_Sensor controls Cooling_Command`". We know from the two triples that if we want to change `Cooling_Command`, we have to change `Zone_Temperature_Setpoint`. `Zone_Temperature_Sensor` is not considered as it is a sensor that cannot be controlled arbitrarily.

While the exact mathematical relationships are not described, dependencies represented in the `controls` relations give enough structure for causal analysis. For example, to analyze if a `Cooling Command` point is working propertly, an application can find related points using the `controls` relationships and analyze data for those points to search for anomalies. If needed, the Brick model can be extended to incorporate more detailed control characteristics such as exact math equations. For example, Ploennigs et al. model linear time invariant dependencies for fault diagnosis [80].

Modeling of control processes is an example of how Brick chooses an appropriate level of abstraction for the task of writing portable energy efficiency applications: Brick models the structure and composition of systems and processes, but not the actual dynamics of those processes.

3.6 Summary

This chapter has presented the design principles of the Brick metadata schema that enable portable applications:

- **Vocabulary Extensibility**: The structure of tags and tagsets allow easy extensions of tagsets for newly discovered domains and devices while allowing inferences of the unknown tagsets with tags.
- **Usability**: Brick represents an entity as a whole instead of as the sum of its annotations. This promotes consistent usage across different actors. Furthermore, the hierarchical class structure lets user queries be more generally applicable across different systems.
- **Expressiveness**: Brick standardizes canonical and usable relationships, which generalize to many different families of applications and can be easily extended with further

specifications.

Brick advances the state-of-the-art by formalizing the semantics of its data model, which enable expressive power with clean extensibility necessary for the construction of portable applications. This is in constrast to the Haystack model and BMS point tags, which rely upon idiom and convention for correct interpretation. Chapter 4 provides the formal construction of the Brick schema using the semantic web model.

	Entities	Occupancy Modeling [52]	Energy Apportionment [51]	Web Displays [10]	Model Predictive Control [99]	Participatory Feedback [60]	Fault Detection and Diagnosis [90]	NILM [66]	Demand Response [109]
Points	Temp Sensor	X					X		
	CO2 Sensor	X							
	Occ Sensor	X	X			X			
	Lux Sensor		X			X			
	Power Meter	X	X	X		X		X	X
	Airflow Sensor			X					
Equipment	*Generic*							X	X
	HVAC	X	X	X			X		
	Lighting	X	X			X			
	Reheat Valve			X			X		
	VAV			X	X				
	AHU				X		X		
	Chilled Water			X			X		
	Hot Water			X			X		
Locations	Building				X		X		
	Floor				X	X		X	
	Room	X	X	X	X	X		X	
	HVAC Zone	X		X	X				
	Lighting Zone	X				X			
Relationships	Sensor `isLocIn` Loc.	X	X			X		X	
	Equip `isLocIn` Loc.		X			X		X	X
	Loc. `hasPart` Loc.		X		X	X			
	Loc. `hasPoint` Sensor	X	X			X		X	X
	Equip `hasPoint` Sensor	X		X			X	X	X
	Equip `hasPart` Sensor			X			X	X	X
	Equip `feeds` Zone	X			X	X			
	Equip `feeds` Room	X			X			X	
	Equip `feeds` Equip			X	X			X	
	Zone `hasPart` Room	X			X	X			

Table 3.2: This table shows at a high level which entities and relationships are required by each of the eight representative applications.

Chapter 4

Representating and Manipulating Brick Models

We have established the abstract design of Brick as a directed node-edge graph whose edges are termed *relationships* and whose nodes are termed *entities*. This chapter presents how Brick implements this design with an ontology, represented with the RDF data model: entities have types that are specified in a class hierarchy, and relationships have constraints on what classes they can apply to. This is followed by a discussion of the requirements of a query language for Brick, and how the SPARQL query language fulfills these requirements.

4.1 Semantic Web Representation

We can reason about the necessary features for a query language and database for Brick using the representation of Brick models as directed, labeled graphs. Any representation of a Brick model therefore needs to capture the set of entities in a Brick graph and all the relationships between them. Applications also need to be able to express *queries* against the Brick model; these queries are executed by a *query processor*. To execute queries, the query processor needs to be able to reason about the structure of the Brick model and its properties given by the Brick ontology. The Brick ontology and Brick models are expressed using existing the *semantic web* [15]. This allows Brick to leverage the set of preexisting ontologies and tools for manipulating, storing and manipulating ontologies.

Definition 4.1: Triple

A triple is a 3-tuple $\langle$`subject`, `predicate`, `object`$\rangle$ expressing that some *subject* has some relationship *predicate* to some other entity *object*. In a graph model, the subject and object are both nodes (Brick entities) and the predicate is a directed edge (Brick relationship).

The semantic web represents knowledge as a directed, labeled graph expressed in terms of length-3 tuples called *triples*. The Brick ontology (comprising the class hierarchy, relationships and their definitions and restrictions) and all Brick models are expressed as a collection of such triples.

Using Semantic Web Ontologies

A crucial feature of the semantic web is it allows ontologies to refer to each other and use relationships, entities, and concepts defined by other ontologies. Brick makes use of several existing ontologies. The base ontology is the Resource Description Framework (RDF) ontology [63], which defines the subject-predicate-object data model and allows Brick models to declare their constituating entities as instances of Brick classes. RDF is extended with the Resource Description Framework Schema (RDFS) [86] and Web Ontology Language (OWL) [76] ontologies. RDFS helps Brick define class hierarchies via super-concepts and sub-concepts, and OWL helps Brick define properties and restrictions on their usage. The RDF data model enables the composition of different kinds of information in buildings such as hierarchical location information (e.g., room-101 is a part of the first floor) and interconnected equipment (e.g., a VAV is fed by an AHU).

An ontology can also imply relationships that are not explicitly expressed in the model, and can influence a query processor's interpretation of relationships. One example is how Brick uses the OWL ontology to define the `feeds` relationship as transitive. To execute queries involving transitive relationships, the query processor needs to be able to determine the set of entities reachable from a specific entity where the entities and relationships involved are subject to a set of query constraints. This involves determining sets of entities separated by an arbitrary number of edges. For example, in Figure 4.1, a query processor needs to be able to find all nodes that are one or more `feeds` relationships away from the `AHU1A` air handling unit node: `VAV2-4`, `VAV-2-3` and `VAV2-3Zone`. The specific set of functionality required of the query processor is detailed in Section 4.2.

Serializing Brick Models

Brick models and Brick ontology must be serialized for storage and manipulation by existing RDF tooling. In the RDF data model and through most RDF tooling, entities and relationships are identified by an IRI (Internationalized Resource Identifier)[1].

An IRI has two components: a *namespace* and a *value*. A *namespace* is a common label for a group of IRIs. Most ontologies are defined within a single namespace, but ontologies may span multiple namespaces in order to make their components more modular. The entities in a Brick model for a particular building also reside in their own namespace; the relationships in these models can refer to other ontologies. A *value* is the name of an entity within the namespace. Entity IRIs can be written fully elaborated in which the namespace

[1]An IRI is a generalization of a URI

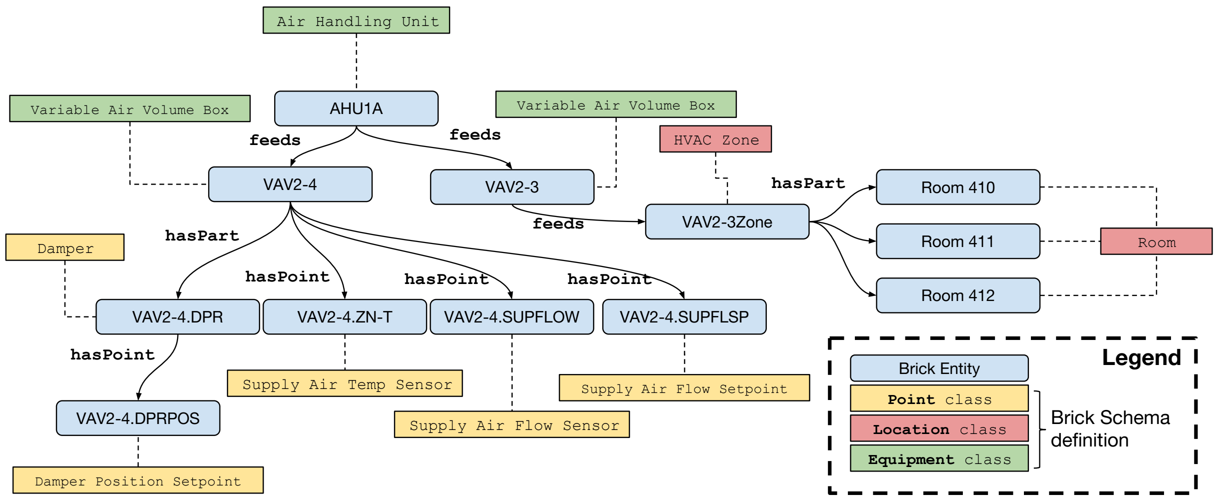


Figure 4.1: A sample Brick model representing a simple HVAC subsystem. Dashed lines indicate the class of each Brick entity. The model captures the structure of the subsystem – the set of equipment and their points of monitoring and actuation – and the process – how the sequence of equipment conditions the air flowing into a set of rooms.

```
@prefix rdf:        <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix brick:      <http://brickschema.org/schema/Brick#> .
@prefix building:   <http://example.com/building#> .

building:myVAV          rdf:type        brick:VAV .
building:myTempSensor   rdf:type        brick:Air_Temperature_Sensor .
building:myVAV          brick:hasPoint  building:myTempSensor
```

Figure 4.2: RDF triples instantiating a VAV and a Temperature Sensor and declaring that the VAV measures temperature via that sensor. Here, `building:myVAV`, `building:myTempSensor` and `building:myVAV` are all Brick entities in the `http://example.com/building> namespace.`

and value are separated with a # character (e.g. `http://brickschema.org/schema/Brick#Temperature_Sensor`) or in a terser form using an abbreviation for the namespace (e.g. `brick:Temperature_Sensor`). Entities with the same name can be distinguished if they are in different namespaces; this is how models can refer to entities defined in different namespaces. A table of common namespaces used in this thesis can be found in Table 4.1.

There are many ways to serialize an RDF data model; one of the more common formats is Turtle [106]. Figure 4.2 contains the Turtle-serialized triples representing the association of an air temperature sensor to a VAV. The first 3 lines of this Turtle file define the mapping from namespace abbreviations to their full IRIs. Turtle serializes subject-predicate-object triples from left to right; on line 5 of Figure 4.2, `building:myVAV` is the subject entity,

Prefix	Namespace
`rdf:`	`http://www.w3.org/1999/02/22-rdf-syntax-ns`
`rdfs:`	`http://www.w3.org/2000/01/rdf-schema`
`owl:`	`http://www.w3.org/2002/07/owl`
`bf:`	`https://brickschema.org/schema/1.0.3/BrickFrame`
`brick:`	`https://brickschema.org/schema/1.0.3/Brick`

Table 4.1: Common namespaces and prefixes

`rdf:type` is the predicate relationships, and `brick:VAV` is the object entity.

Line 5 declares an entity identified by the label `building:myVAV`, this creates the `myVAV` entity in the `building` namespace. `brick:VAV` is a TagSet defined by Brick representing the variable air-volume box class; `rdf:type` declares `building:myVAV` to be an instance of `brick:VAV`. Similarly, line 6 instantiates an `Air_Temperature_Sensor` with the label `building:myTempSensor`. Line 7 uses the Brick relationship `brick:hasPoint` to declare that `building:myVAV` is functionally associated with the given temperature sensor. This example also demonstrates how entities from multiple namespaces can be used together in the same document, or even the same triple.

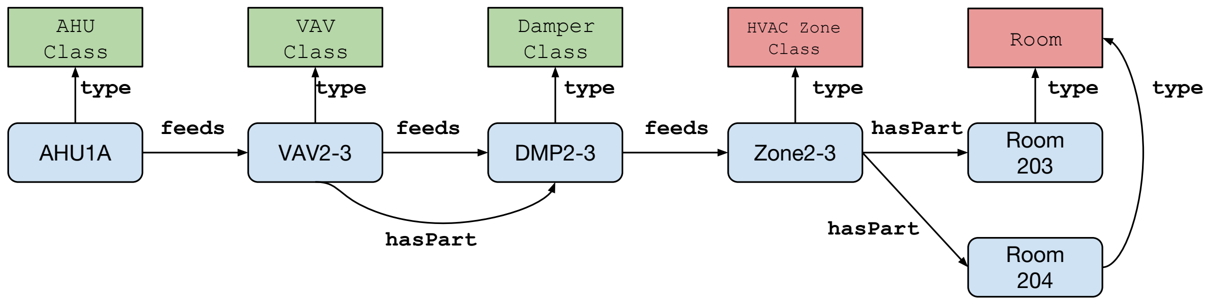
4.2 Requirements of a BRICK Query Language

Recall that Brick's goal is to provide a common semantic representation of buildings that enables building portable applications, which are applications that can execute against multiple buildings without substantial reconfiguration. Applications achieve portability through their ability to extract from *any* Brick model the set of entities and relationships that are useful to the execution of that application, using one or more *queries* against that Brick model. Queries are executed against a Brick model through the use of a *query processor*. In order for queries to be portable, they need to account for differences in building as they are represented in Brick models. Brick models differ in three primary ways:

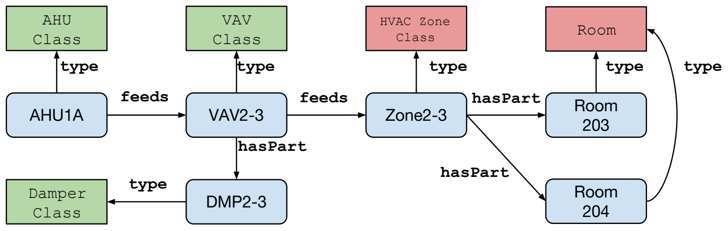
- in how entities are classified (e.g., is this sensor a `Temperature Sensor`, `Air Temperature Sensor` or a `Supply Air Temperature Sensor`?),
- in how entities are related (e.g., how complete is the Brick model's representation of all the equipment involved in a cold water or hot water loop?), and
- in the ground truth of the model (e.g., does the building even contain the features that interest a specific application?).

To address these differences, an effective query language needs to be able to express what is known *abstractly*. This is implemented through the following mechanisms:

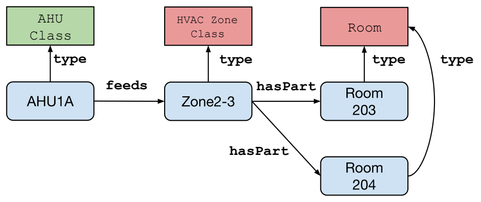
Firstly, the query language needs to be able to query both the Brick ontology and Brick model. This allows queries to perform introspection into the Brick ontology. One common



(a) Brick model of an HVAC process showing air flowing from an AHU through a VAV, the VAV's damper, and finally into the HVAC zone.



(b) Brick model of an HVAC process showing air flowing from an AHU through a VAV and into the HVAC zone. The damper is not considered part of the flow.



(c) Brick model of an HVAC process only showing the association from an AHU to an HVAC zone.

Figure 4.3

application of introspection removes the need to know *a priori* the classes used in a particular Brick model; instead, an application author can express a query over the Brick ontology and a Brick model to find which subclasses have been instantiated in the Brick model from any point in the class hierarchy. For example, rather than asking which instances of the Brick `Room` class are in a Brick model, an application query can ask which instances of any subclass of the Brick `Room` class are in a Brick model. This is possible because the Brick ontology and class hierarchy can be queried in the same manner as the Brick model.

Secondly, the query language needs to be able to find collections of entities connected by arbitrary sequences of relationships. Brick queries often need to find collections of related entities for which the functional nature of that relationship is known – entities that are part of the same process flow, or measure one another, or are contained within one another – but the specific structure of those entities and relationships in a particular Brick model is unknown.

The last necessary feature of an effective Brick query language is the ability to include optional components in the query.

It is important to note that these mechanisms for introducing flexibility into the Brick query model do not undercut the value of idioms – informal patterns for expressing common

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX brick: <http://brickschema.org/schema/Brick#>
SELECT ?ahu ?room
WHERE {
    ?zone rdf:type brick:HVAC_Zone .
    ?room rdf:type/rdfs:subClassOf* brick:Room .
    ?ahu rdf:type/rdfs:subClassOf* brick:AHU .
    ?ahu brick:feeds+ ?zone .
    ?zone brick:hasPart ?room .
}
```

Figure 4.4: A simple SPARQL query for retrieving all rooms connected to a given Air Handling Unit (AHU). (Duplicate of Figure 4.7)

building substructures in Brick. Idioms make Brick easier to use, but are not necessary for interpretation. For example, sometimes differences of opinion can arise in how to model a particular scenario: consider Figure 4.3a and Figure 4.3b, which illustrate the two Brick models for whether a damper is included in the sequence of equipment for conditioned air flow. Idioms can help resolve these scenarios, but importantly, it is always possible to reason about a Brick model because all entities in a Brick model (even non-idiomatic Brick models) are tied to the Brick ontology. This means that a Brick model is always well-defined.

Example

To elucidate these requirements, consider an application that wants to know which rooms in a building are served from a particular AHU unit. Figure 4.3 depicts three different Brick models representing different interpretations of the same simple HVAC system in which an AHU passes air to a VAV, which in turn passes the conditioned air on to a collection of two rooms called an HVAC zone These different representations may arise from differences in the availability of information (i.e., does the underlying BMS expose information about the VAV's damper). The edges labeled `feeds` indicate that there is air flowing from one entity to another; this indicates the direction, or "flow," of the HVAC process. The `hasPart` edge indicates that one entity is composed of another entity: an HVAC zone consists of rooms, and a VAV consists of its subcomponents such as the indicated damper.

Brick models 4.3a and 4.3c differ in completeness (4.3c elides the VAV equipment and its damper) and Brick models 4.3a and 4.3b differ in their interpretation of the process. 4.3a considers a VAV's damper a first class citizen of the HVAC process that provides conditioned air to the HVAC zone whereas 4.3b does not. Such differences in completeness and interpretation should be expected, especially from site to site. The question therefore is how an application can find the information it needs in such a situation.

The application expresses the types of the entities it wants in its queries. These types can be determined by an application author; this particular application needs to find instances of the `AHU`, `HVAC Zone` and `Room` classes. To write the *portable* query that relates these concepts, the application author leverages Brick's relationships: air handling units and HVAC zones will be connected by the `feeds` relationship because the HVAC zone is the recipient of conditioned air flowing from an air handling unit. In order for the application query to be portable across many Brick models, it cannot make too many assumptions about the structure of a Brick model. Concretely, it cannot account for how many pieces of equipment will be expressed between an air handling unit and an HVAC zone across the array of Brick models against which the application may run. By using the query language's ability to express entities connected by arbitrary sequences of relationships, the application author can remain *agnostic* to the particular sequence expressed in a particular Brick model. In this example, the query would account for this variation by expressing that the air handling unit and HVAC zone are connected by one or more `feeds` relationships.

The portable query for this application is written in Figure 4.4; lines 7 and 8 demonstrate an example of performing introspection across the Brick ontology to discover which relevant classes have been instantiated in the Brick model this query is executing against. Line 9 demonstrates matching arbitrary sequences of the `feeds` relationship. Because this query uses the portability features of the Brick query language, it will execute correctly over each of the three representations in Figure 4.3. The syntax of the query language will be explained in Section 4.3.

4.3 The SPARQL Query Language

The W3C recommends the use of SPARQL (SPARQL Protocol and RDF Query Language) [96] for querying the RDF data model. SPARQL provides a set of analytic query operations that can be expressed over graph patterns and graph traversals. [6] shows that SPARQL has the same expressive power as relational algebra under bag semantics (in which duplicates are not ommitted from the result set). This expressive power comes at the cost of increased complexity of evaluation, which impacts query time. This cost is discussed in Chapter 8).

The requirements for a query language for Brick can be fulfilled with a subset of SPARQL's features (see Table 4.2). The required SPARQL features will be explained and motivated through the course of this section.

Most Brick queries are SPARQL SELECT queries. SELECT queries return a subset of the contents (entities and relationships) of an RDF graph, such as a Brick model, in the form of a table. The "columns" of this table are the names of the variables contained in the SELECT clause. The contents of the table are the result of evaluating a SPARQL WHERE clause over the contents of the underlying RDF graph. A SPARQL WHERE clause consists of variables, triple patterns and subqueries.

WHERE clauses contain one or more triple patterns, which are linked together with shared variables: one triple pattern might specify that the `?thermostat` variable is an instance of Brick's `Thermostat` class, and another might specify that the `?thermostat` variable

Query Forms		Clauses and Operators		Property Paths	
Name	Brick?	Name	Brick?	Name	Brick?
`SELECT`	**yes**	`WHERE`	**yes**	Inverse Path (`^pred`)	no
`CONSTRUCT`	no	`OPTIONAL`	**yes**	Sequence (`pred1 / pred2`)	**yes**
`ASK`	no	`UNION`	**yes**	Alternative (`pred1 | pred2`)	**yes**
`DESCRIBE`	no	`ORDER BY`	no	Zero or more (`pred1*`)	**yes**
		`BOUND`	no	One or more (`pred+`)	**yes**
		`FILTER`	no	Zero or one (`pred?`)	**yes**

Table 4.2: The set of features of the SPARQL 1.1 query language required by Brick.

is an entity that has a temperature sensor. A row in the results table corresponds to an assignment of variables in the SELECT to entities in the RDF graph for which the triples constituting the WHERE clause exists in the RDF graph.

Definition 4.2: SPARQL variable

Variables are labels for entities that a query wants to resolve. Variables are denoted with a `?` prefix, e.g. `?thermostat`. The name of the variable has nothing to do with its meaning; a variable called `?thermostat` does not need to resolve to an instance of Brick's `Thermostat` class.

Definition 4.3: triple pattern

A triple pattern is an RDF triple (subject-predicate-object) where at least one of its terms is a variable. Triple patterns describe graph traversals. An example of a triple pattern is

```
?thermostat rdf:type brick:Thermostat
```

A triple pattern's predicate can relate its subject and object using arbitrary sequences of predicates. There are a few common methods that Brick queries use:

- Exact sequences: a triple pattern can describe entities connected by an exact sequence of edges by delimiting a list of predicates with the `/` character, e.g. `rdf:type/rdfs:subClassOf` is an edge with label `rdf:type` followed by an edge with label `rdfs:subClassOf`. The `/` operator can be thought of eliding an intermediate variable, such as in Figure 4.6.

```
SELECT ?damper ?vav WHERE {
    ?damper     rdf:type/rdfs:subClassOf*   brick:Damper .
    ?vav        rdf:type/rdfs:subClassOf*   brick:VAV .
    { ?damper   brick:isPartOf      ?vav    }
    UNION
    { ?damper   brick:isFedBy       ?vav    }
}
```

Figure 4.5: An example of the SPARQL `UNION` operator, which can express multiple subqueries.

- Arbitrary sequences: a triple pattern can describe entities connected by sequences edges of one or more types, using the `+` (one or more) or `*` (zero or more) or `?` (zero or one) suffixes, e.g. `bf:feeds+` is one or more edges that all have the `bf:feeds` label.

SPARQL WHERE clauses can also contain `UNION` and `OPTIONAL` elements, which define additional triple patterns that may be used to filter the graph. These are important features for SPARQL as applied to Brick models because they allow an additional degree of flexibility that cannot be captured by the class hierarchy. The class hierarchy can only describe entities more generally (by moving up the hierarchy) or more specifically (by moving down the hierarchy); sometimes, queries need to be able to describe one or more possible groups of entities that may not have a common ancestor in the class hierarchy, or may be related in more than one way.

The `UNION` and `OPTIONAL` operators allow SPARQL to express these "branches" of query logic. Triple patterns in a WHERE clause have an "and" relationship: each triple pattern places further constraints on the results of the query. The `UNION` operator specifies one or more alternative patterns (called *subqueries*) to be evaluated as part of the query. The results of `UNION` subqueries have an "or" relationship. The `OPTIONAL` operator specifies a subquery which can extend the results returned by a query if information matching the subquery exists. An illustrative scenario is a query against the Brick model in Figure 4.3a: a user may want to ask if a damper is related to a VAV through either the `feeds` relationship *or* the `hasPart` relationship. In SPARQL, this would use the `UNION` operator (Figure 4.5).

SPARQL Idioms for Brick

A few idioms have emerged for Brick models expressed in RDF and queried with SPARQL.

A natural way of interacting with the Brick class hierarchy and instances of those classes is through the paradigm of *subtype polymorphism* (also called *subtyping*), which is a major class of polymorphism used by languages like C++. The Brick class hierarchy relates subclasses to their superclasses through the `rdfs:subClassOf` predicate. `rdfs:subClassOf` is a transitive property, so sequences of `rdfs:subClassOf` convey further subclassing.

This mechanism is used by queries such as those that want to relate entities to classes in the Brick class hierarchy; for example, all instances of subclasses of the Brick `Temperature`

```
SELECT ?sensor WHERE {
    ?sensor     rdf:type/rdfs:subClassOf*       brick:Temperature_Sensor
}

# alternative expression, without the / concatenation operator
SELECT ?sensor WHERE {
    ?sensor           rdf:type           ?immediate_class .
    ?immediate_class rdfs:subClassOf*   brick:Temperature_Sensor
}
```

Figure 4.6: Simple SPARQL query demonstrating the `rdf:type/rdfs:subClassOf*` idiom for finding instances of arbitrary subclasses of a Brick parent class (`brick:Temperature_Sensor`).

`Sensor` class. Queries can leverage SPARQL's matching of arbitrary edge sequences to express this idea in a composite predicate: `rdf:type/rdfs:subClassOf*`. Observe Figure 4.6 as an example: the `rdf:type` predicate captures that the entities resolved for `?sensor` should be an instance of some class. The `rdfs:subClassOf*` predicate captures that that class is either `brick:Temperature_Sensor` (for which the `*` pattern matches zero edges) or some subclass of `brick:Temperature_Sensor` (for which the `*` pattern matches one or more edges).

Another common idiom in Brick models is querying sequences of equipment in process flows, such as a hot or cold water system. This uses the compound predicate `bf:feeds+`, which matches one or more sequences of the `bf:feeds` relationship.

Idioms provide users with a set of abstractions for common query tasks. The query mechanisms used to implement these idioms can also guide optimizations in the query processor. Chapter 7 explores the impact of Brick idioms and SPARQL features on the performance of SPARQL query processors.

SPARQL Versions

Several crucial features of SPARQL that fulfill the Brick query language requirements are part of the SPARQL 1.1 specification [47], rather than the base SPARQL 1.0 specification. The primary feature is *property paths*[2], which can specify arbitrary path lengths between two entities in a graph. The reliance upon SPARQL 1.1 has significant impact on the requirements of the query processor (Chapter 7).

Example: Querying a Brick model With SPARQL

Figure 4.7 is a query for retrieving all rooms which are connected to a given AHU. Lines 1-3 declare the prefixes for the various namespaces to shorten the references to entities; for

[2]`https://www.w3.org/TR/sparql11-query/#propertypaths`

brevity, we omit these from all later queries in this paper. Line 4 contains the `SELECT` clause, which states that the variables `?ahu` and `?room` should be returned (the `?` prefix indicates a variable). The `WHERE` clause determines the types and constraints on these variables.

```
 1 PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 3 PREFIX brick: <http://brickschema.org/schema/Brick#>
 4 SELECT ?ahu ?room
 5 WHERE {
 6     ?ahu    rdf:type/rdfs:subClassOf*   brick:AHU .
 7     ?zone   rdf:type/rdfs:subClassOf*   brick:HVAC_Zone .
 8     ?room   rdf:type/rdfs:subClassOf*   brick:Room .
 9     ?ahu    brick:feeds+    ?zone .
10     ?zone   brick:hasPart   ?room .
11 }
```

Figure 4.7: A simple SPARQL query for retrieving all rooms connected to a given Air Handling Unit (AHU).

A building represented in Brick can specify the specific subclasses, or if that information is not available, instantiate a generic class. Line 6 is a common construct in Brick queries which accounts for the uncertainty in which class of AHU has been instantiated in the queried Brick model. This triple pattern returns all entities `?ahu` that are either an instance of a subclass of `brick:AHU` or an instance of `brick:AHU` itself. An application that does not require specific features of such subclasses may want to query for the generic class rather than exhaustively specify every possible subclass. Lines 7 and 8 serve a similar purpose for the `brick:HVAC Zone` and `brick:Room` classes.

After declaring the types of the entities involved, the query restricts the set of relationships between the entities on lines 9 and 10 to determine which pairs of entities are connected. Line 9 finds all HVAC zones downstream of a particular AHU by following a chain of `brick:feeds` relationships (the `+` indicates that 1 or more edges can be traversed as long as the edges are of type `brick:feeds`). Line 10 links the identified HVAC zones with the rooms they contain. The correct relationships to use can be determined from the Brick relationship list (Table 3.1).

This example query illustrates an important quality of Brick queries: establishing a link between two entities (even across different subsystems such as HVAC and spatial) does not require explicit knowledge of all intermediary entities. Rather, the query denotes the relevant entities and relationships: the query in Figure 4.7 is indifferent to whatever building-specific equipment and details lie between an Air Handler Unit and the end zones. This is possible because the relationships between those entities all use Brick's `brick:feeds` relationship.

4.4 Summary

This chapter has presented an implementation of the Brick schema's formal semantics as an ontology using the RDF data model. The Brick ontology is built on several existing ontologies – RDF, RDFS and OWL – which provide the formal mechanisms for defining the Brick class hierarchy and relationships. This formalism is what enables Brick to accurately and completely describe the structures and processes within many different buildings, which makes Brick suitable for implementing portable applications.

This chapter also establishes the requirements of a Brick query language that enables portable applications. Brick queries retrieve the information required for an application's configuration and operation. These requirements can be fulfilled by a subset of the features offered by the SPARQL 1.1 query language.

Chapter 5

Creating Brick Models

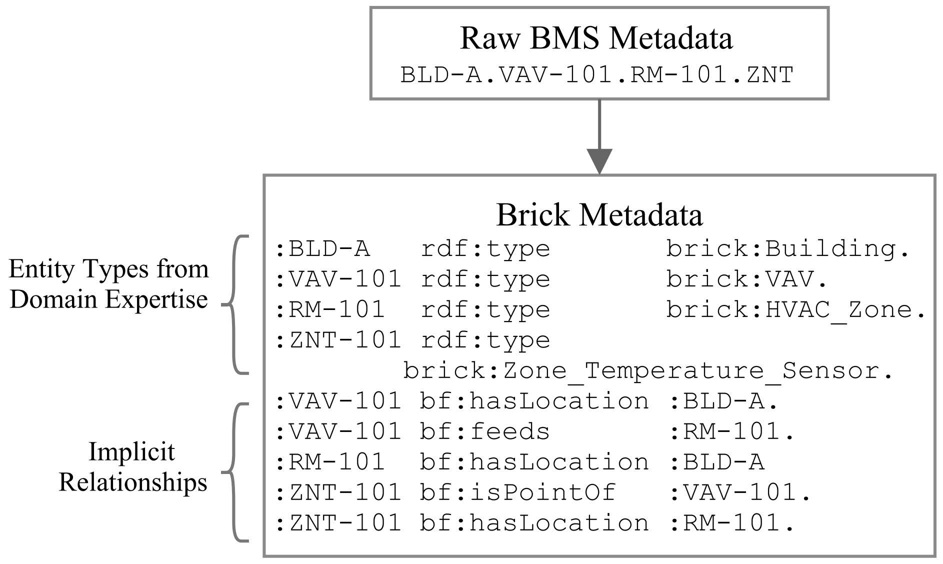
It is possible to extract Brick entities and relationships for a partial Brick model from existing metadata representations of buildings which use conventions and idioms of the trade to overcome a lack of formal semantics. We describe a general approach for converting building metadata to Brick, present initial conversion techniques for the popular Haystack and IFC building schemata, and demonstrate these techniques on three real buildings. Finally, we describe methods to convert vendor specific BMS metadata to Brick.

The general approach is to parse the given building metadata into sets of entities that have obvious relationships between them and then add these entities and relationships to a Brick model; from there, the Brick model can be refined. The success of a conversion depends on what information is captured in a schema and how that information is structured. For unstructured metadata, the conversion implementation is often site-specific. For structured metadata, the conversion implementation is more portable.

5.1 Converting BMS Points to Brick

Converting BMS metadata to Brick requires extracting the semantic information, i.e. entities and relationships, from these point names. Because point names are inconsistently named across (and within) buildings, domain experts – individuals familiar with the BMS and the building subsystems – need to provide the deconstruction of point names into the set of underlying types (Brick classes) and labels (Brick entities).

Figure 5.1 shows an example of a BMS point and the equivalent Brick representation. This single BMS point contains several idiomatic abbreviations for entities that Brick makes explicit and distinct: the `ZNT` component indicates that the point is a `Zone Temperature Sensor`. The `RM-101` component means that this temperature sensor measures the air in room number `101`; there is not enough information to tell if the sensor is physically located inside the room, or if it is in a duct supplying conditioned air to the room. The `VAV-101` component of the point name indicates that the conditioned air flowing into the room is from a variable air volume box, and `BLDG-A` names which building contains this whole structure.



```
BLD-A.VAV-101.RM-101.ZNT
```

```
:BLD-A   rdf:type       brick:Building.
:VAV-101 rdf:type       brick:VAV.
:RM-101  rdf:type       brick:HVAC_Zone.
:ZNT-101 rdf:type
        brick:Zone_Temperature_Sensor.
:VAV-101 bf:hasLocation :BLD-A.
:VAV-101 bf:feeds       :RM-101.
:RM-101  bf:hasLocation :BLD-A
:ZNT-101 bf:isPointOf   :VAV-101.
:ZNT-101 bf:hasLocation :RM-101.
```

Figure 5.1: An example of mapping raw metadata of a BMS point to Brick. The abbreviations inside the raw metadata represent some entities in the BMS and the mapping can be given by a domain expert or inferred by an automated inference algorithm. The relationships between entities in the raw metadata are implicit.

The result of decoding substrings into Brick entities and Brick classes yields the set of RDF triples in Figure 5.1 labeled as “Entity Types from Domain Expertise.”

Extracting entities from point labels requires an understanding of the naming conventions of the particular building or BMS. Once the entities are extracted, it is possible infer the relationships between them using the structure of Brick’s ontology. The ontology informs the set of possible relationships between entities with a certain type: for example, a zone temperature sensor (a Brick `Point`) and a VAV (a Brick `Equipment`) most likely have a `isPointOf` relationship between them. This process yields the set of RDF triples in Figure 5.1 labeled “Implicit Relationships.”

However, BMS point names do not always contain all the information necessary to fully populate a Brick model, such as which VAVs are downstream of a particular AHU. This information could be obtained through interviews with building managers or control engineers, or also through data-driven techniques that perturb BMS control points to find links between equipment [82]. For the six case study buildings described in Chapter 6, we generate Brick instances using a semi-automated script per building which parses point names to generate Brick entities and infer the relationships between them[1]. The results of this conversion are summarized in Table 6.1 in Chapter 6.

[1]`https://github.com/BuildSysUniformMetadata/Brick`

BMS Point Normalization Frameworks

There are several available frameworks that reduce the effort of converting BMS metadata to standardized vocabularies (such as Brick) through automation. Most of the frameworks focus on identifying point types, but all produce structured representations that are simple to convert to Brick. Methodologies include clustering BMS metadata on syntactic similarity of point names to reduce the number of classes that must be identified by a building domain expert [11][50]. Gao et al. extract features from time series data to learn models for point types [42]. Pritoni et al. propose learning the relationships between AHUs and VAVs by observing reactions of devices to artificial perturbations [82]; this corresponds directly to Brick's `feeds` relationship. Bhattacharya et al. propose a framework to construct synthesis rules from examples presented to building managers and domain experts [16]. The synthesis rules extract all possible relationships in BMS metadata which usually covers Equipment, Point, Location and relationships among them like `hasLocation` and `isPartOf`. Scrabble [59] expands on the methodology established in [16], using active learning and transfer learning to reduce the number of classification samples that need to be given to a domain expert. Plaster [58] is a complementary benchmarking framework that allows different metadata normalization methods to be more directly compared.

5.2 Converting Haystack to Brick

While Project Haystack's data model does not capture all of the entities and relationships that Brick can express, it does provide a more structured view into the equipment and points for a building than idiosyncratic BMS labels. A Project Haystack building model can be used to bootstrap a more detailed Brick model. To this end, we have developed a simple Haystack to Brick converter. The converter has two components: a translator module that maps Haystack entities to Brick TagSets, and a relationship module that infers a possible set of Brick relationships between those entities using contextual information and a set of basic assumptions.

Methodology

Recall that Haystack entities refer to equipment, sensors, setpoints and other physical objects and are described using a combination of marker tags, value tags and references to other entities. The task of the translator module is to determine a mapping from sets of Haystack tags to a single Brick TagSet. For each Haystack entity, the translator finds the Brick TagSet whose tags have the largest intersection with the entity's Haystack tags. The translator then adds an instance of that Brick TagSet to the output Brick model corresponding to the Haystack entity. Because Brick's tags are based off of Haystack's tag dictionary, this technique is able to correctly translate the majority of Haystack entities in the Haystack models we have observed.

Haystack's tags do not explicitly label all of the information contained in a Haystack model. As a result, sometimes additional entities and relationships can be extracted from parsing the *values* of Haystack value tags. The most common case of this involves Haystack's lack of spatial tags. Haystack does not formally represent HVAC zones – a logical grouping of physical space conditioned by a single terminal unit – or rooms or other spaces as entities in a Haystack model. Instead, zones are implicit in the definition of a VAV entity with a `zone` tag. What is lost is not just the name of the zone (although this is usually one-to-one with a VAV entity), but also the collection of physical spaces associated with that zone. As a result, users of Haystack typically encode the name of rooms or zones in the name of the VAV entity, just like in a BMS (Figure 5.2). In these cases, the translator module must be programmed with how to extract room- or zone-level information from the names of Haystack entities. This is analagous to extracting Brick metadata from BMS labels.

The relationship module uses value tags associated with Haystack entities to populate the set of relationships around the translated Brick entity. Some of these tag-value pairs describe aspects of the entity such as engineering units or square footage, which could be captured in future Brick extensions. Other tag-value pairs use "-Ref" tags to relate entities (e.g. `equipRef`, `siteRef`, `elecMeterRef`). These references do not capture the full set of relationships required by Brick, but can be unambiguously mapped to a specific Brick relationship: The `elecMeterRef` value tag implies a Brick `hasPoint` relationship between an equipment entity and an electric meter entity. The `siteRef` value tag asserts which building (or site) an entity is associated with. The `ahuRef` value tag indicates which air handling unit is upstream of a terminal unit, which corresponds to the Brick `isFedBy` relationship. Many Haystack entities also use the `equipRef` tag, which has the following definition:

> Association with an equip entity. When used on a point this indicates a sensor/cmd/setpoint associated with the equipment. When used on an equip it indicates nesting/containment. ([83])

This definition translates to Brick as follows: if the owner (the entity that has the tag) and the target (the value of the `equipRef` tag) are both equipment, then we infer a Brick `feeds` relationship between the entities; if the owner is a sensor and the target is equipment, then we infer a Brick `isPointOf` relationship. With these simple contextual assumptions, the Haystack identifiers of the owner and target of a "-Ref" tag are enough to generate the requisite Brick triples.

Haystack-to-Brick Example

Figure 5.2 contains the Haystack representation of a VAV in Ghausi Hall on the UC Davis campus. This example illustrates the use of several idiosyncratic tags used by the authors of the Haystack model to encode information beyond what Haystack formally defines:

```
area: "546 sq ft"
associatedRooms: "3165, 3240"
canopyHoodSignage: true
code: "VAV-1D"
dcv: 1.0
done: true
equip: true
equipParent: "AHU 04"
equipRef: "AHU 04"
id: "VAV 4_16 Rm 3167"
navName: "VAV 4_16 Rm 3167"
priorityTwo: true
siteRef: "Ghausi"
vav: true
```

Figure 5.2: Haystack VAV entity, with spatial information encoded in the entity identifier string. Note that only underlined tags are standardized in Project Haystack. The original author of this metadata needed to add the other tags.

```
Ghausi:AHU_04               rdf:type        brick:Air_Handling_Unit .
Ghausi:VAV_4_16             rdf:type        brick:VAV .
Ghausi:Room3167             rdf:type        brick:Room .
Ghausi:Room3165             rdf:type        brick:Room .
Ghausi:Room3240             rdf:type        brick:Room .
Ghausi:HVAC_Zone_4_16       rdf:type        brick:HVAC_Zone .
Ghausi:HVAC_Zone_4_16       bf:hasPart      Ghausi:Room3167 .
Ghausi:HVAC_Zone_4_16       bf:hasPart      Ghausi:Room3165 .
Ghausi:HVAC_Zone_4_16       bf:hasPart      Ghausi:Room3240 .
Ghausi:AHU_04               bf:feeds        Ghausi:VAV_4_16 .
Ghausi:VAV_4_16             bf:feeds        Ghausi:HVAC_Zone_4_16 .
```

Figure 5.3: The Brick triples (entities and relationships) generated from the Haystack entity in Figure 5.2.

- The nonstandard `associatedRooms` tag defines the set of rooms served by the VAV; note that because Haystack has no list data type, the authors of the Haystack model encoded the set of rooms as a comma-delimited string.
- The room number in the entity identifier string ("VAV 4_16 Rm 3167") indicates a third room associated with this VAV, but it is unclear from the representation what the relationship of this room is to the rooms captured by the `associatedRooms` tag.

After the translator has been informed about these local conventions, it can derive instances of three rooms, an AHU, a VAV and an HVAC zone from this Haystack entity. The relationship module can then identify the Brick relationships between them, resulting in the triples contained in Figure 5.3.

We have implemented our Haystack-to-Brick converter script in Python, totaling 350 lines of code[2]. We apply the converter to two Haystack models from the UC Davis campus and were able to successfully translate air handling units, VAVs, dampers, HVAC zones, rooms, setpoints and electric meters as well as temperature, humidity and occupancy sensors. Ghausi Hall is a 66,000 sq ft engineering building with 2,183 Haystack entities; the translated Brick model contains 4,135 triples. PES is a 90,000 sq ft office and lab building with 6,475 Haystack entities; the translated Brick model contains 15,561 triples.

5.3 Converting IFC to Brick

The IFC building information model captures more detailed construction-oriented structural information for buildings than Brick, but it is possible to generate partial Brick models from IFC representations of buildings. IFC models mostly consist of spatial information useful for construction such as the size and position of walls, dampers and ducts, but also include semantic groupings of these entities into floors, rooms and HVAC zones. The IFC schema encodes information as “objects”, which correspond to equipment, spaces, other infrastructure and groups of objects.

We have implemented a simple converter that exports spatial information in IFC models to Brick. The converter first scans an IFC model for all instances of `IFCZONE` objects, which can correspond to an HVAC Zone, and `IFCSPACE` objects, which correspond to rooms. `IFCRELASSIGNSTOGROUP` objects associate zones (using a “RelatingGroup” attribute) with a list of rooms (using a “RelatedObjects” attribute). `IFCRELAGGREGATES` objects associate rooms with floors (instances of `IFCBUILDINGSTOREY`).

Our IFC-to-Brick converter, implemented in 100 lines[3] of Python (not including an open-source IFC file parser[4]), converts IFC representations of floor, room and zones to their Brick equivalents. The converter currently makes the assumption that all zones are HVAC zones because there is not enough contextual information in the IFC model to determine the “kind” of zone without programatically traversing the components of the HVAC system as represented in the IFC model. We have successfully tested the converter on an IFC model of a 7,000 sq ft office building in downtown Berkeley. The textual IFC model totals some 150,000 lines and the exported Brick model contains 159 triples. This informally illustrates the expressive differences between IFC and Brick; the IFC model contains a very detailed description of the construction physical space, but the translated Brick model only represents the high-level spatial information required by building applications.

Refer to Lange, et al [62] for a more comprehensive study of the Brick entities and relationships that can be derived from an IFC model, a survey of real-world IFC models, and a performant framework for extracting Brick-relevant information from IFC models.

[2]`https://github.com/gtfierro/BrickConvert/tree/master/haystack`

[3]`https://github.com/gtfierro/BrickConvert/tree/master/ifc`

[4]`https://github.com/mvaerle/python-ifc`

5.4 BrickMason Framework

The above sources of metadata are not always available. Residential and small/medium commercial buildings for instance lack the pre-assembled (albeit unstructured and nonstandard) metadata necessary for the management of large commercial buildings with established building management systems, but are also much simpler to describe. These classes of buildings are increasingly controlled and monitored by networked, consumer-facing "Internet of Things" (IoT) devices such as the Nest thermostat[5], Pelican thermostat[6], Enlighted lighting system[7] and Rainforest Eagle[8], whose digital interfaces present an opportunity to extract contextual information about the building in which they are deployed.

We have built BrickMason[9], an extensible framework for collecting and coordinating the metadata exposed by heteroeneous and multi-vendor systems that manage buildings into a unified Brick model. To create a Brick model for a site, a user configures a set of BrickMason plugins and then executes BrickMason. A BrickMason plugin is a Python module that extracts metadata from an external source and inserts it into a Brick model. BrickMason plugins execute sequentially on a shared Brick model, each adding new entities and relationships or refining existing ones.

Disambiguating labels is the primary challenge for the integration of independent metadata sources: different sources might refer to the same Brick entities, but under different names. Additionally, different sources may present incorrect or incomplete information. For example, BrickMason may extract the names of HVAC zones from a smart thermostat scheduling system and the names of rooms from a meeting room scheduling system, but may not have a source that ties HVAC zones to rooms. A third source, such as a survey with a building manager or an examination of a building's floor plan, could provide the association of HVAC zones to rooms, but might refer to HVAC zones by the model number of their corresponding rooftop unit, rather than the human-assigned name from the smart thermostat scheduling system.

BrickMason allows plugins to implement their own disambiguation techniques. As each plugin executes, it can query the partially-generated Brick model for similarly named or prexisting entities, and choose whether or not to align its own derived metadata with what is already expressed in the Brick model. BrickMason currently employs very simple methods for disambiguation – string similarity measures and regular expressions – but could be integrated with a framework like Scrabble [59] for more robust matching. Table 5.1 contains a list of some of the implemented BrickMason plugins.

[5]`https://web.archive.org/web/20181202012548/https://nest.com/thermostats/`

[6]`https://web.archive.org/web/20181202012641/https://www.pelicanwireless.com/`

[7]`https://web.archive.org/web/20180420135521/https://www.enlightedinc.com/system-and-solutions/iot-applications/light/`

[8]`https://web.archive.org/web/20181202013043/https://rainforestautomation.com/rfa-z114-eagle-200/`

[9]`https://web.archive.org/web/20181202013504/https://github.com/gtfierro/BrickMason`

Plugin Name	Data Source	Metadata Added
`haystack`	Haystack Models	HVAC equipment and zones, metering, sensors and relationships
`ifc`	IFC Models	HVAC zones, rooms, floors
`nws`	National Weather Service API	weather stations
`revit`	Revit Models	thermostats, rooms, floors, HVAC zones, sensors, lights, metering and relationships
`enlighted`	Enlighted lighting system API	lights, lighting zones, rooms, occupancy sensors
`pelican`	Pelican thermostat system API	thermostats, HVAC zones

Table 5.1: A subset of available BrickMason plugins that coordinate to produce a single Brick model for a site.

5.5 Summary

The ease of creating a useful Brick model is crucial to the utility and adoption of Brick. Though the first Brick models were constructed by experts in a semi-automated fashion, this chapter has demonstrated how Brick models can be created by normalizing semantic information from pre-existing sources of digital metadata. These sources explicitly and implicitly refer to the entities and relationships represented by Brick. Multiple sources can be fused together into a single Brick model; this requires reconciling differences in naming and representation.

The synthesis and verification of Brick models is an active area of research. Approaches range from applying timeseries classification [50] to supervised machine learning [16, 59] to automated transformation of semi-structured metadata [62]. These development efforts, together with growing commercial interest in Brick, may produce more robust and easy-to-use tools for automated conversion of Brick models, and may even lead to the use of Brick during the design and construction process.

Chapter 6

Evaluation of Brick Expressiveness

The contents of the Brick ontology and class hierarchy are driven by an empirical study of the concepts already captured by existing BMS as well as the concepts that building applications wanted to use. In this chapter, we evaluate the expressiveness, effectiveness and completeness of Brick.

6.1 Empirical Method

Recall that Brick seeks to create a robust, queryable representation of a building's equipment, subsystems, points and relationships that is *complete* (it can express and name necessary entities), *expressive* (it can express and name necessary relationships) and *usable* (it can represent entities and relationships in a way that supports the development of portable building applications). This chapter evaluates Brick along these dimensions on a set of six buildings.

The set of real-world buildings was created as part of the collaborative effort from sixteen researchers spanning seven institutions across the U.S. and Europe to develop the original release of Brick [9]. Together, this team compiled BMS ground-truth metadata from six buildings to bootstrap the Brick schema development. The ground-truth metadata consists of equipment and point labels extracted from the BMS for each building and human-curated documentation of the *type* of each label (Figure 6.1). We then developed a consistent naming scheme for each of the identified types using a standardized set of tags inspired by Haystack, and then organized these types into a class hierarchy consistent with their root types (Figure 3.2). We filled in missing classes from the hierarchy. For example, a `Discharge Air Pressure Sensor` implies the existance of a more generic `Pressure Sensor`; even though it is unlikely that any BMS would label a point solely as a "pressure sensor", it is still a useful classification that aids in discovering available types of sensors.

This methodology attempts to avoid the danger of *overfitting*, in which the emerging class hierarchy becomse too tightly tailored to the entities of particular building used to develop it and loses the ability to generalize its classes to other buildings. To reduce overfitting,

```
SODA4R731__ASO  =>  R731 -> zone; SOD -> site; A4 -> ahu; ASO -> temp setpoint off;
SODA2S14___SMK  =>  S14 -> supply fan; SMK -> smoke alarm; SOD -> site; A2 -> ahu;
SODA1S11___MAT  =>  S11 -> supply fan; SOD -> site; MAT -> mixed air temp; A1 -> ahu;
SODA3R315_RVAV  =>  RVAV -> vav reheat discharge air pressure sensor; R315 -> zone;
                          SOD -> site; A3 -> ahu;
SODA3R723__ASO  =>  R723 -> zone; SOD -> site; A3 -> ahu; ASO -> temp setpoint off;
SODA3R327__AGN  =>  R327 -> zone; AGN -> pid loop gain; A3 -> ahu; SOD -> site;
SODH1P02___FLT  =>  H1 -> hot water loop; SOD -> site; P02 -> pump; FLT -> fault sensor;
SODA3R798__ART  =>  ART -> zone temp sensor; R798 -> zone; SOD -> site; A3 -> ahu;
SODA1R405B_ARS  =>  R405B -> zone; SOD -> site; A1 -> ahu; ARS -> zone temp setpoint;
SODA3R683_RVAV  =>  RVAV -> vav reheat discharge air pressure sensor; R683 -> zone;
                          SOD -> site; A3 -> ahu;
SODA1R405B_ART  =>  ART -> zone temp sensor; R405B -> zone; SOD -> site; A1 -> ahu;
SODA3R311__AGN => R311 -> zone; AGN -> pid loop gain; A3 -> ahu; SOD -> site;
```

Figure 6.1: Snippet of ground truth for Soda Hall on the UC Berkeley campus. The decomposition of the BMS labels on the left into the components on the right was performed automatically by the tool from [16].

we first created a draft class hierarchy using the points found in four of the six buildings and evaluated the coverage (% of points that can be appropriately classified using Brick) on the last two buildings: Soda Hall and Rice Hall. We then incorporated points from Soda and Rice Hall into the draft class hierarchy. The point coverage of all six buildings for the resulting Brick class hierarchy (corresponding to the *completeness* of Brick) is in Table 6.1. This established class hierarchy, together with the ontology-defined relationships, constitutes the first release of the Brick schema.

To evalute the *expressiveness* and *usability* of Brick, we execute a family of eight representative applications against a Brick model corresponding to each of the six case study buildings. We observe how well SPARQL queries express the entities and relationships constituting the desired context for an application and whether or not the SPARQL queries can recover the necessary information when evaluated against each Brick model. The list of applications and the result of the applications is in Table 6.2.

6.2 Case Study Buildings

Here, we review the construction and control system for each of the six buildings, and discuss the challenges faced in producing the Brick model for each site. Table 6.1 contains an overview of the six buildings.

Gates Hillman Center at CMU

The Gates and Hillman Center (GHC) at Carnegie Mellon University is a relatively new building, completed in 2009, with 217,000 square feet of floor space, 9 floors, and 350+ rooms of various types (offices, conference rooms, labs). It contains over 8,000 BMS data

Building Name	**Location**	**Year**	**Size (ft^2)**	**# Points Points**	**% TagSets Mapped**	**# Relationships Mapped**
Gates Hillman Center (GHC)	Carnegie Mellon Univ., Pittsburgh, PA	2009	217,000	8,292	99%	35,693
Rice Hall	Univ. of Virginia, Charlottesville, VA	2011	100,000	1,300	98.5%	2,158
Engineering Building Unit 3B (EBU3B)	UC San Diego, San Diego, CA	2004	150,000	4,594	96%	8,383
Green Tech House (GTH)	Vejle, Denmark	2014	38,000	956	98.8%	19,086
IBM Research Living Lab	Dublin, Ireland	2011	15,000	2,154	99%	14,074
Soda Hall	UC Berkeley, Berkeley, CA	1994	110,565	1,586	98.7%	1,939

Table 6.1: Case Study Buildings Information.

points for HVAC sensors, setpoints, alarms, and commands. CMU contracts with Automated Logic[1] for building management.

The GHC includes 11 AHUs of different sizes serving multiple zones: three small AHUs serve a giant auditorium, a big laboratory and three individual rooms respectively. Eight large AHUs supply air to more than 300 VAVs. GHC's HVAC system also contains computer room air conditioning (CRAC) systems which are equipped with additional cooling capacity to maintain the low temperature in a computer room and fan coil units systems to provide cooling and ventilation functions. Brick matched 99% of GHC's BMS points, with the remaining points being too uncommon to be required by most applications (such as a `Return Air Grains Sensor` which measures the mass of water in air). The direct translation of BMS tags into Brick was relatively simple, only requiring a mapping between the human-readable BMS data points and Brick for each unique data point type. This process was helped by the fact that the BMS points provided by Automated Logic were mostly human readable, allowing the mapping process to proceed more quickly. That said, there were a number of BMS tags which we found to be unintelligible, and in these cases we spent considerable time determining the meaning of these points by examining Automated Logic documentation.

The major challenge in converting the GHC to Brick was determining the relationships between pieces of equipment not encoded in the BMS labels. While the information is available through an Automated Logic GUI representation of the building, there was no machine readable encoding of which VAVs connected to which AHUs. This required examining the building plans directly to incorporate more than 400 relationships. The Brick representation, on the other hand, is both machine- and human-readable.

Rice Hall at UVA

Rice Hall hosts the Computer Science Department at the University of Virginia. The building consists of more than 120 rooms including faculty offices, teaching and research labs, study

[1]Automated Logic, `http://www.automatedlogic.com/`

areas and conference rooms distributed over 6 floors with more than 100,000 square feet of floor space. The building contracts with Trane[2] for building management.

Rice Hall contains four AHUs associated with more than 30 Fan Coil Units (FCU) and 120 VAVs serving the entire building. Besides the conventional HVAC components, the building features several different new air cooling units, including low temperature chilled beams and ice tank-based chilling towers, an enthalpy wheel heat recovery system, and a thermal storage system. The building also contains a smart lighting system including motorized shades, abundant daylight sensors and motion sensors. Rice Hall's BMS points are easily interpretable for conversion to Brick despite it containing some uncommon equipment such as a heat recovery and thermal storage systems as part of the building design as an energy-efficient "living laboratory". Brick's class structure was extended to incorporate BMS points unique to Rice Hall among the other case study buildings such as `Ice Tank Entering Water Temperature Sensor`. Additionally, Brick's relationships were able to properly capture how these uncommon pieces of equipment were connected to other parts of the building subsystems.

Engineering Building Unit 3B at UCSD

The Engineering Building Unit 3B (EBU3B) at University of California, San Diego hosts the Department of Computer Science & Engineering and contains offices, conference rooms, research laboratories, an auditorium and a computer room. The building was constructed in 2004 and has 150,000 square feet of floor space with over 450 rooms. The BMS of EBU3B is provided by Johnson Controls[3], and contains more than 4500 data points, most of which related to the HVAC system and power metering infrastructure.

The HVAC system consists of a single AHU that supplies conditioned air to 200+ VAV units and some FCUs. There are exhaust fans for all kitchens and restrooms and a CRAC system serving the computer room. The HVAC system also has Variable Frequency Drives (VFD), valves, heat exchangers and cooling coils to facilitate operation of the AHU and CRAC. The Brick schema provides the necessary TagSets and relationships for all of these components and their monitoring and control points. The university central power plant provides the hot and cold water for domestic medium temperature water system and controlling air temperature in the HVAC. The corresponding sensors that measure the hot and cold water use such as flow rate and temperature were modeled in Brick, but the central plant was left out as it was not part of the building. The building contains meters measuring power consumption of various subsystems: lighting, computer room, HVAC system and elevators.

An issue in mapping EBU3B to Brick is that the AHU discharge air is divided into two parts for two wings of the building. Brick currently does not model how the discharge air in the AHU is divided into two wings but describes the connections to other equipment such

[2]Trane, `https://www.trane.com/`

[3]Johnson Controls, `http://www.johnsoncontrols.com/`

as VAVs. Additionally, EBU3B's BMS contains data points related to Demand Response (DR) events such as load shedding for hot water, which exposes an interesting conflation of the representation and operation of the building. Because BMSes have been typically written as monolithic applications over vendor-specific interfaces, they must incorporate external signals such as DR into the set of BMS points directly. On the other hand, Brick decouples the resources and infrastructure of a building from the building operation so that any application can operate on top of Brick representation.

Soda Hall at UC Berkeley

Soda Hall, constructed in 1994, houses the Computer Science Department at UC Berkeley. It mostly consists of closed small to medium sized offices, where either faculty or groups of graduate students sit. The BMS system, provided by the now-defunct Barrington Systems, exposes only the data points in the HVAC system.

The HVAC system of the building runs on pneumatic controls, and comprises 232 thermal zones. The zones on the periphery of the building have VAVs with reheat, while the other zones do not. Each zone has a VAV; VAVs for the zones on the periphery of the building have reheat mechanism. For a VAV with reheat, the same control setpoint indicates both the amount of reheat and the amount of air flowing into a zone.While such combination is building-specific, Brick can express the fact that the same sensor controls both the reheat and air flow by duplicating the point and labeling the copies with the `Reheat Command` and `Air Flow Setpoint` TagSets. The logic of the setpoint also can be described with control relationships in Brick for dependencies to other setpoints related to actual reheat and air flow rate. The logic for communicating with the point correctly would be handled by some other system; Brick simply identifies the available points.

Unique to the other buildings presented here, the operational set of Soda Hall's HVAC components is not static. Soda Hall contains a redundant configuration of chillers, condensers and cooling towers. At any point of time, one of these systems is operational while the others are kept as standby. An isolation valve setpoint indicates which of the redundant subsystems is currently operating. Brick completely expressed the redundant subsystem arrangement, but the equipment contained several unique points such as `On Timer` for the chiller subsystem that had to be added to Brick's TagSets.

Green Tech House

The Green Tech House (GTH) was constructed in 2014 as a 38,000 square feet office building in Vejle, Denmark. It contains 50 rooms spanning three stories and functions as office spaces, a cafeteria, meeting rooms and bathrooms. GTH is controlled by the Niagara BMS[4], but to protect basic building functionality only a subset of the BMS points are exposed via oBIX.

[4]Tridium, `https://www.tridium.com/`

As the oBIX points do not include AHU nor VAV points, the Brick representation was constructed from a combination of BMS points, BMS screenshots and technical documents.

Compared to the rest of the case study buildings, the thermal conditioning of GTH is reversed: Air is heated centrally in a single AHU and distributed to VAVs with cooling capabilities. The AHU uses a rotary heat exchanger to recovers heat from the return air. The pressure of the AHU return and supply air for the north and south side of the building is measured separately. Additionally, most rooms have radial heating on either walls or in the floor. These are supplied by two independent hot water loops – one for wall-mounted heaters and one for floor heaters – heated by district heating.

IBM Research Living Lab

The IBM Research building in Dublin was retrofitted as modern 15,000 m^2 office in 2011 from an old factory. The building serves as living laboratory for IBM's Cognitive Building research and is heavily equipped with modern building automation technology to provide a rich data source for research.

The building has been renovated multiple times and new systems were installed by different companies. The heterogeneity of systems became very high in the building. The building contains 2,154 points collected from 11 different systems. The building is served by 4 AHUs with 115 points but also has old disconnected legacy systems in the point list. Unlike the other buildings, it contains 250 smart meters and 150 desk temperature sensors. It has 1,000 points for 161 fan coil units (FCUs) as well as 350 points on the lighting system including 150 PIR sensors and door with people counters.

The configuration of the FCUs connected to different AHU, boilers and chillers are unique for this building while terminal units such as VAVs and FCUs are connected to a single central unit such as an AHU in the other buildings. It shows importance of the relationship modeling and the capability of Brick.

6.3 Model Coverage of Buildings

We evaluate the *completeness* of Brick by measuring how many entities Brick can describe for a set of real-world buildings (point coverage). The draft class hierarchy (before the last two buildings were incorporated) was able to match 93.5% of Rice Hall's BMS points and 93.1% of Soda Hall's BMS points, demonstrating that the draft class hierarchy sufficiently captured the diversity of data points available. After incorporating the BMS points from Rice Hall and Soda Hall into the class hierarchy, the point coverage improved to 98.5% of Rice Hall's BMS points and 93.7% of Soda Hall's BMS points.

The point coverage of each building on the completed Brick class hierarchy (as of the first Brick release) is contained in Table 6.1 under the "% TagSets" column. Brick is able to match at least 96% of points exposed by each building's BMS. The unclassified points are part of the "long tail" characteristic of the types of points in buildings [17], and consists of a few rare and unlabeled points as well as undocumented, propreitary BMS points that are not used in applications.

	Building					
Application	EBU3B	GTH	GHC	IBM	Rice	Soda
Occupancy [52]	261	245	366	821	265	232
Energy Apportionment [51]	-	302	-	397	4	-
Web Displays [10]	699	81	65	835	106	605
MPC [99]	482	69	428	324	110	482
Participatory Feedback [60]	-	253	-	386	-	-
FDD [90]	229	29	229	728	-	136
NILM [66]	6	82	-	1348	-	-
Demand Response [109]	2300	24	2490	608	4	152

Table 6.2: Number of matching rows in each building for the SPARQL queries consisting the eight applications. A non-zero number indicates that the application successfully ran on the building. Buildings with '-' did not have any relevant points exposed in the BMS.

6.4 Query Coverage of Buildings

We evaluate the *expressiveness* and *usability* of Brick by implementing application queries for eight representative building applications – one from each of the application categories compiled by Bhattacharya et al. [17] – and observing how those queries are expressed and to what extent they extract the application-relavant information from each of the real-world buildings. The applications themselves are described in Chapter 3.

Application Coverage

We implemented each of the applications in Table 3.2 in a building-agnostic manner as a set of SPARQL queries. These SPARQL queries are listed in Appendix A. The evaluation of the SPARQL queries yields the set of entities and relationships corresponding to the features of the application. Table 6.2 shows shows how many rows were returned when running each application's queries against each of the case study buildings. If a table cell contains a -, then the application queries did not return any results, meaning the application could not run.

The results demonstrate that the Brick ontology is sufficiently expressive to model the application requirements as portable SPARQL queries. The viability of executing applications across the case study buildings was instead limited by the information presented by the respective BMS and by the lack of digital control systems for building subsystems (such as lighting).

Here we review the performance of each application as implemented through a family of SPARQL queries.

- **Occupancy Modeling Application**: this application requires occupancy information in order to operate. The application was able to run across all 6 case study buildings because many HVAC systems expose an occupancy indicator for each VAV (and thus each HVAC Zone). Room-level occupancy information is much rarer, but the application queries are written to take advantage of it if it exists.

- **Energy Apportionment Application**: the implemented application relies upon lighting subsystem state being reported in the BMS. Because this is more sparsely available than HVAC system information, the application was only able to run on 3 of the 6 case study buildings.

- **Web Displays Application**: the software thermostat implemented by this application uses VAV reheat and cooling valve status and control, which is commonly available in BMS. The application optionally leverages power meter data to provide users with a gauge of how their thermal comfort requests impact energy consumption. The case study buildings do not have floor or room-level submetering, so the application had to use full-building or HVAC system metering. The application query is able to express this flexibility.

- **Model-Predictive Control Application**: this application utilizes spatial information as well as AHU and VAV status and command information that is commonly available in BMS. As a result, it is able to execute across all of the case study buildings.

- **Participatory Feedback Application**: the implemented application relies upon lighting information, which is only available for 2 of the case study buildlings

- **Fault Detection and Diagnosis Application**: the implemented application contains a suite of similar FDD techniques that can be implemented from available HVAC system information. The application queries encode this flexibility with generous use of the SPARQL `UNION` operator.

- **Non-Intrusive Load Monitoring Application**: the implemented application only needs power meter data in order to run and as a result the SPARQL queries find results for all buildings that expose meter data in the BMS. What is not captured by Brick in the application queries are requirements involving the sample rate of the building meter – NILM techniques work best when the sample rate is higher.

```
SELECT ?airflow_sensor ?room ?vav
WHERE {
  ?airflow_sensor rdf:type/rdfs:subClassOf* brick:Supply_Air_Flow_Sensor .
  ?vav rdf:type brick:VAV .
  ?room rdf:type brick:Room .
  ?zone rdf:type brick:HVAC_Zone .
  ?vav brick:feeds+ ?zone .
  ?room brick:isPartOf ?zone .
  ?airflow_sensor brick:isPointOf ?vav .
}
```

Figure 6.2: Genie query for airflow sensors and rooms for VAVs. The query returns all relevant triples for Genie to bootstrap itself to a new building.

- **Demand Response Application**: because the application only implements simple load shedding, any degree of HVAC control is relevant to its operation. Because this information is commonly exposed in BMS, the application is able to execute across all of the case study buildings.

The primary challenge in developing portable queries was accounting for the variance in relationships across buildings. For example, a zone temperature sensor may have an `isPointOf` relationship with an HVAC zone or a VAV. These inconsistencies arise from differences in building construction and the representation of the points in the BMS. However, it is possible to account for these differences in SPARQL to construct truly portable queries.

Example Application: Genie

We show an example application from the perspective of Brick. The Genie [10] application incorporates monitoring and modeling of HVAC zone behavior and power usage with occupant feedback to provide a platform for occupants to directly contribute to the efficacy and efficiency of a building's HVAC system. Genie requires the following relationships:

- the mapping of VAVs to HVAC zones and rooms
- the heating and cooling state of all VAVs in the building
- the mapping of VAV airflow sensors to rooms
- all available power meters for heating or cooling equipment

Immediately, the requirements of this application outstrip the features provided by other metadata solutions. Genie needs to relate entities across subsystems typically isolated or ignored in modern BMS: the spatial construction of the building, the functional construction of the HVAC system, and the positioning of power meters in that infrastructure. Brick simplifies this cross-domain integration and makes it possible to retrieve all relevant information in a few simple queries.

To identify the airflow sensors and rooms served for each VAV, the application uses the query in Figure 6.2. Lines 3-4, 5, 6, 7 find all the `Supply Air Flow Sensor`s, `VAV`s, `Room`s and `HVAC Zone`s in the building respectively. Line 8 identifies the `VAV`s that feed the respective `HVAC Zone`s and line 9 identifies the `Room`s that are part of the corresponding `HVAC Zone`s. Line 10 finds the `Supply Air Flow Sensor`s that are part of the corresponding `VAV`s. The application uses Brick's synonyms to capture both `Discharge Air Flow Sensor`s as well as `Supply Air Flow Sensor`s. The "Web Displays" row of Table 6.2 contains the results of running Genie over the six buildings.

6.5 Reflections

In order to evaluate the expressiveness and completeness of Brick, we implemented a family of representative applications to be executed against each of the case study buildings. These properties are measured by the proportion of entities and relationships the applications want to refer to that can also be expressed in Brick. The implementation of these applications as queries and the execution of these queries against the case study buildings indicates that Brick is able to capture all of the application requirements. The viability of executing the applications against each of the case study buildings is limited by the availability of information in each buildings BMS, rather than the expressive power of Brick.

Chapter 7

Basis for Workload-driven Design of a Brick Query Engine

Initial evaluation (Chapter 6) determined that RDF/SPARQL fulfill Brick's requirements of description and representation. This chapter examines the question of how well suited these technologies are to fulfilling the "systems" requirements of Brick queries integrated into building applications. We address three questions regarding this integration:

1. What are the characteristics of the Brick workload, and what requirements does the workload place on a Brick query processor?
2. How well do existing RDF/SPARQL databases meet these requirements?
3. If they do not, how can we leverage the characteristics of the Brick workload to design a query processor that does meet these requirements?

We focus on evaluating whether existing RDF/SPARQL technology supports latency-sensitive applications including user interfaces, building modeling, demand response, alarms and model-predictive control. We target a query response time of <100ms, a conventional interactive latency threshold [73]. This chapter presents a performance evaluation of several popular RDF databases against the Brick workload, represented by seven Brick queries of varying complexity on three real Brick building models. We then characterize the Brick workload by the graph properties of Brick models and the required query language features. Finally, we use these findings to develop HodDB, a RDF/SPARQL query processor for Brick that consistently meets the latency demands of Brick applications (Chapter 8).

7.1 RDF Database Overview

RDF databases – sometimes called "triplestores" – are specialized graph databases that store and manipulate triples. These systems provide the storage of collections of RDF triples, each constituting a graph, and the retrieval of stored data through a query language such as SPARQL. Due to the requirements of a Brick query language, our evaluation focuses on available RDF databases that implement the SPARQL query language. This disqualifies several other RDF and graph databases (such as Cayley [24], Dgraph [33], Badwolf [43] and Neo4j [71]), which implement alternative graph query languages such as Gremlin [88] and Cypher. An evaluation of other query languages and databases is a subject for future work.

We evaluate Brick workload performance on six SPARQL query processors: three open-source RDF databases, an open-source Python library, and two closed-source RDF databases:

Apache Jena [102] is an open-source Java framework for managing and querying RDF data. It contains a web frontend (Fuseki) and a SPARQL backend (TDB) that supports all SPARQL 1.1 features. TDB maps URIs to short, numerical ids and stores these in YARS-style B-tree indices [48] (explained below), which is a common implementation approach.

Blazegraph [100, 101] is a commercial, open-source graph database capable of storing up to 50 billion RDF triples on a single machine, but also supports distributed storage. It provides a full SPARQL 1.1 implementation, with support for transactions based on MVCC for write-heavy workloads. Blazegraph also uses YARS-style indices with internal numerical identifiers inserted into B+-trees, which is similar to Jena. Blazegraph also supports geospatial data.

RDF-3X [72] is an unmaintained open-source RDF database that uses compressed YARS-style indices. RDF-3X was developed before SPARQL 1.1, and does not support any of the property path operators from Table 7.6.

RDFLib [104] is an open-source Python module for storing and querying RDF graphs. It provides a full SPARQL 1.1 implementation on top of B-tree indices, and does not explicitly optimize for large-scale datasets, choosing to focus on feature-completeness. We use the Sleepycat persistence engine shipped with RDFLib, which is backed by BerkeleyDB.

Allegrograph [1, 38] is an ACID-compliant, commercial, closed-source graph database for storing billions of RDF triples. It provides a full SPARQL 1.1 implementation in addition to support for geospatial and temporal data.

Virtuoso [36, 95] is a commercial database that provides support for RDF and SPARQL over a relational database, rather than the B-tree indices typical of the other RDF databases. Virtuoso supports full SPARQL 1.1.

This is not an exhaustive set of RDF databases, but all are prevalent in the literature and available for download. Noted omissions are TopBraid Live [105], for which we could not obtain an evaluation copy, and the performant RDF extension [65] to the FastBit [113] storage system, which has no available implementation.

7.2 RDF Database Benchmark Methodology

We evaluate the performance of several popular SPARQL databases on three Brick graphs using a set of seven queries used by real Brick applications requiring low and predictable latency. These queries are different than the set of queries used to evaluate the expressiveness

```
### VAV Enum (Building Dashboard)
SELECT ?vav WHERE {
  ?vav rdf:type brick:VAV .
}

### Temp Sensors (Building Dashboard, Room Diagnostics)
SELECT ?sensor WHERE {
  ?sensor rdf:type/rdfs:subClassOf* brick:Zone_Temperature_Sensor .
}

### AHU Children (Building Dashboard)
SELECT ?x WHERE {
  ?ahu rdf:type brick:AHU .
  ?ahu bf:feeds+ ?x .
}

### Spatial Mapping (Building Dashboard)
SELECT ?floor ?room ?zone WHERE {
  ?floor rdf:type brick:Floor .
  ?room rdf:type brick:Room .
  ?zone rdf:type brick:HVAC_Zone .
  ?room bf:isPartOf+ ?floor .
  ?room bf:isPartOf+ ?zone .
}

### VAV Relships (Building Dashboard)
SELECT ?vav ?x ?y ?z ?a ?b WHERE {
  ?vav rdf:type brick:VAV .
  ?vav bf:feeds+ ?x .
  ?vav bf:isFedBy+ ?y .
  ?vav bf:hasPoint+ ?z .
  ?vav bf:hasPart+ ?a .
}
```

Figure 7.1: The first set of SPARQL queries used in real-world Brick apps, used here for benchmarking RDF databases in §7.2.

and completness of Brick in Chapters 3 and 6. The queries presented here instead exercise the range of query language features required by Brick.

Experimental Setup: Brick Queries and Applications

Figures 7.1 and 7.2 show the set of representative queries used for benchmarking. All queries are drawn from the Brick apps described above.

`VAVEnum` is a simple enumeration of all VAVs in a building. This is a trivial query intended to measure the base performance of a SPARQL query processor. This flavor of query (list all instances of this type) is a very common interaction with Brick graphs; nearly all Brick queries involve a clause of this form.

`TempSensors` finds all sensors that are instances of zone temperature sensors or any subclass thereof. This is a more advanced, but still common, form of the `VAVEnum` query

```
### Sensors In Rooms (Room Diagnostics)
SELECT ?sensor ?room
WHERE {
  { ?sensor rdf:type/rdfs:subClassOf* brick:Zone_Temperature_Sensor . }
  UNION
  { ?sensor rdf:type/rdfs:subClassOf* brick:Discharge_Air_Temperature_Sensor . }
  UNION
  { ?sensor rdf:type/rdfs:subClassOf* brick:Occupancy_Sensor . }
  UNION
  { ?sensor rdf:type/rdfs:subClassOf* brick:CO2_Sensor . }
  ?vav rdf:type brick:VAV .
  ?zone rdf:type brick:HVAC_Zone .
  ?room rdf:type brick:Room .
  ?vav bf:feeds+ ?zone .
  ?zone bf:hasPart ?room .
  {?sensor bf:isPointOf ?vav }
  UNION
  {?sensor bf:isPointOf ?room }
}

### Grey Box (Automatic Grey Box Modeler)
SELECT ?vav ?room ?temp_uuid ?valve_uuid ?setpoint_uuid WHERE {
  ?vav rdf:type brick:VAV .
  ?vav bf:hasPoint ?tempsensor .
  ?tempsensor rdf:type/rdfs:subClassOf* brick:Temperature_Sensor .
  ?tempsensor bf:uuid ?temp_uuid .
  ?vav bf:hasPoint ?valvesensor .
  ?valvesensor rdf:type/rdfs:subClassOf* brick:Valve_Command .
  ?valvesensor bf:uuid ?valve_uuid .
  ?vav bf:hasPoint ?setpoint .
  ?setpoint rdf:type/rdfs:subClassOf* brick:Zone_Temperature_Setpoint .
  ?setpoint bf:uuid ?setpoint_uuid .
  ?room rdf:type brick:Room .
  ?tempsensor bf:hasLocation ?room .
}
```

Figure 7.2: The second set of SPARQL queries used in real-world Brick apps, used here for benchmarking RDF databases in §7.2.

that uses both the / and * property path operators. The challenge when evaluating a query such as `TempSensors` is the need to traverse an arbitrarily large number of edges (here, edges of the type `rdfs:subClassOf`).

`AHUChildren` lists all equipment and sensors downstream of an air handler unit. This query is similar in structure to `TempSensors`, but uses the + property path operator instead.

`SpatialMapping` associates floors, the rooms on that floor, and the HVAC zones that cover those rooms. This query makes use of the + property path operator in order to avoid any assumptions about the exact associations between floors, rooms and HVAC zones (i.e. a room could have a `bf:isPartOf` relationship with a logical grouping such as a department or company, which in turn has a `bf:isPartOf` relationship with a `brick:Floor` instance).

`SensorsInRooms` associates a family of sensors with a room, using the room's HVAC zone and VAV information. The query makes heavy use of `UNION` to select the appropriate sensor classes.

`VAVRelships` finds the set of "things" related to a VAV: whats upstream and downstream of it, what measurement points it has, and what equipment it contains. This query is expensive to evaluate because it resolves to a large number of values, resulting in a number of expensive joins.

`GreyBox` identifies, for each room in a building, a minimal set of sensor streams (identified by a UUID) that can be used to train a simple grey box thermal model.

Our evaluation of the Brick workload uses the following latency-sensitive applications:

Building Dashboard queries a Brick model to render a dashboard for different building subsystems. 100ms is a common target for users to feel an interaction is "instantaneous" [73]. The dashboard application requires queries similar in structure to those generated by the interactive query explorer described in §8.4.

Automatic Grey Box Modeler uses a Brick model to formulate a series of simple thermal models trained on HVAC timeseries data. Used in a model-predictive control loop, the response time of the metadata model should be minimal to leave more time for the rest of the computation.

Room Diagnostics monitors the sets of sensors in each room to check for uncomfortable or unsafe conditions (such as high temperatures or CO_2 levels). The app queries the Brick model often to make sure it is using the most up-to-date description of the building, and needs to quickly react to dangerous settings by querying the model for the correct alarms to trigger.

Figure 7.1 and Figure 7.2 show the queries constituting these applications. Other categories of applications that can benefit from fast metadata queries are fast demand response [78], model-predictive control, and online fault detection and diagnosis. [9] and [17] present more comprehensive lists of metadata-driven applications.

Experimental Setup: Brick Models

We evaluate the Brick workload of 7 queries over three buildings: CIEE is a small (7.5k sq ft) office building with a single floor and five rooftop units. It has been retrofitted with an array of wireless sensors as well as networked lighting and thermostats. Soda Hall (110k sq ft, abbreviated as "Soda") and Sutardja Dai Hall (100k sq ft, abbreviated as "SDH") are large buildings with combined office and laboratory space. Both expose sensing and actuation points through a building management system. The graph properties of the Brick models for these buildings are shown in Table 7.5 (discussed later).

Our evaluation consists of running the set of Brick queries against these Brick models using each database, and measuring the distribution of response times. We compare the 99th percentile of this distribution to our target latency bound of 100ms.

We develop a simple test harness[1] to dispatch each benchmark query against each database and measure the time in milliseconds from the time the query was dispatched to the time

[1] `https://github.com/gtfierro/brick_database_eval`

the response is received. The test harness ensures that queries do not run concurrently and that only one Brick graph is loaded into a database at a time. After a simple preprocessing step (described below), the test harness loads a graph into a database and executes a query 200 times. We apply a timeout of 5 minutes to each query; due to pathological cases exposed by the Brick queries, query execution can sometimes take several hours on existing query processors. Before each run of queries, the test harness restarts each database, removes its persistent storage and forces it to reload the dataset to ensure a “cold-start” state for each set of 200 requests.

The test harness has been designed to make our benchmark results reproducible. Each evaluated database has a corresponding Dockerfile [67] for consistent and replicable execution of each database. We will continue to improve and expand the test harness and benchmark suite for evaluating the performance of RDF databases and SPARQL query processors on Brick workloads.

The preprocessing step ensures that all queries run correctly on each database by populating a Brick graph with all inverse edges. Many of the relationships defined in Brick have inverses and either edge can be used in a query even if only one is explicitly defined in the RDF source triples. For example, an AHU having a `bf:feeds` relationship with a VAV could also be expressed as a VAV having a `bf:isFedBy` relationship with an AHU. These inverse relationships are defined in the Brick ontology using standard techniques defined by the OWL ontology [7]. Most of the RDF databases we tested do not implement the necessary inference, so each Brick graph had to be pre-populated with the set of all inverse edges because the queries were not written with knowledge of which of the inverse edges were used in the original definition of the building.

All data was gathered on an server with a 3.5 GHz Intel Xeon E5-1650 CPU; all databases were backed by a dedicated SSD.

7.3 Evaluation of Existing RDF Databases on Brick Workload

Tables 7.1, 7.2 and 7.3 show the mean, standard deviation and 99th percentile latencies for each of the benchmark queries (Figure 7.1 and Figure 7.2) over the three Brick buildings from Table 7.5. We report the distribution for completeness, but 99th percentile latency is the key metric. We defer discussion of the last column (HodDB) until Chapter 8. We begin by drawing some broader conclusions about the data, and then examine specific results to understand how the structure of these databases interacts with the structure of Brick queries and graphs. Figure 7.3 visualizes the benchmark 99^{th} percentile results to draw attention to how well each database meets the performance target (the bold heptagon).

Most databases exhibit good query performance (within the 100ms bound) on the small building (Table 7.1), but substantially degraded performance on the two larger buildings (Tables 7.2 and 7.3). Only Allegrograph, Blazegraph and Virtuoso are able to complete each

Query	Jena			Blazegraph			RDF-3X			RDFLib			Allegrograph			Virtuoso			HodDB		
	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}
`VAVEnum`	11	7	16	19	13	25	5	1	7	9	1	12	8	7	19	4	0	6	4	1	6
`TempSensor`	24	10	43	53	16	61	-	-	-	16	1	18	38	9	47	6	1	8	4	0	6
`AHUChildren`	13	8	21	20	13	24	-	-	-	10	1	13	8	7	19	5	1	7	4	1	6
`SpatialMapping`	20	15	39	66	17	81	-	-	-	**182**	**5**	**198**	66	11	99	8	1	12	4	1	6
`SensorsInRooms`	59	12	93	25	16	49	-	-	-	**330**	**8**	**356**	**156**	**13**	**174**	5	5	7	5	1	8
`VAVRelships`	9	2	14	22	13	32	-	-	-	15	1	18	9	8	20	5	1	7	4	1	6
`GreyBox`	12	7	21	24	16	37	-	-	-	53	5	65	11	10	20	5	2	6	6	1	8

Table 7.1: Query latency distribution for the small building (CIEE). All times are in milliseconds. A `-` denotes the query did not return any results. **Bold** indicates that the 99th percentile latency is outside the 100ms bound.

Query	Jena			Blazegraph			RDF-3X			RDFLib			Allegrograph			Virtuoso			HodDB		
	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}
`VAVEnum`	14	9	27	26	17	51	9	2	14	51	13	83	27	12	55	21	6	38	6	2	10
`TempSensor`	**63**	**29**	**104**	58	20	79	-	-	-	56	14	88	**158**	**23**	**214**	23	8	40	6	1	9
`AHUChildren`	19	15	58	60	22	91	-	-	-	**134**	**17**	**182**	**84**	**20**	**133**	37	10	63	8	2	19
`SpatialMapping`	**5547**	**108**	**5752**	**84**	**19**	**114**	-	-	-	**224981**	**633**	**226782**	**1788**	**67**	**2192**	44	13	76	15	3	23
`SensorsInRooms`	$>$ **5min**			**290**	**47**	**401**	-	-	-	$>$ **5min**			**2206**	**80**	**2460**	**69**	**19**	**112**	31	6	52
`VAVRelships`	**83**	**29**	**152**	**367**	**31**	**432**	-	-	-	**1243**	**33**	**1344**	**4974**	**151**	**5107**	**312**	**27**	**397**	42	10	78
`GreyBox`	**174**	**38**	**239**	**305**	**36**	**380**	-	-	-	$>$ **5min**			**264**	**24**	**341**	**77**	**59**	**116**	38	8	59

Table 7.2: Query latency distribution for a large building (Soda).

Query	Jena			Blazegraph			RDF-3X			RDFLib			Allegrograph			Virtuoso			HodDB		
	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}
`VAVEnum`	8	3	12	23	16	40	6	1	9	26	2	36	15	8	24	11	2	14	5	1	8
`TempSensor`	53	6	73	56	18	79	-	-	-	32	3	45	**91**	**11**	**115**	12	2	16	5	1	9
`AHUChildren`	12	2	16	47	19	68	-	-	-	75	5	93	46	11	59	18	3	14	6	1	8
`SpatialMapping`	**6257**	**78**	**6509**	60	19	88	-	-	-	**58686**	**413**	**59896**	**786**	**46**	**967**	21	3	30	9	2	15
`SensorsInRooms`	$>$ **5min**			**933**	**52**	**1005**	-	-	-	$>$ **5min**			**1213**	**55**	**1256**	30	8	39	10	3	16
`VAVRelships`	19	2	26	**266**	**35**	**357**	-	-	-	**731**	**18**	**807**	**2748**	**107**	**3001**	**193**	**25**	**263**	26	9	61
`GreyBox`	**189**	**73**	**248**	**189**	**47**	**297**	-	-	-	$>$ **5min**			**158**	**19**	**210**	**130**	**75**	**161**	26	6	42

Table 7.3: Query latency distribution for a large building (SDH).

query on the two large Brick buildings in less than 5 minutes[2]. Virtuoso performs closest to the 100ms latency target: its 99th percentile latency fails only on `VAVRelships` and `GreyBox`.

To understand the demands the Brick workload places on a query processor, we examine which query features exhibit poor performance across buildings and databases. Over the suite of queries in Table 7.6, the two primary factors are the number of patterns in a query and use of the `*` and `+` property path operators. The evaluation of each pair of patterns in a SPARQL query that share at least one variable requires a join, as does the traversal of each

[2]In fact, we have observed Jena taking around 7 hours completing the `SpatialMapping` query on a spinning metal drive.

Path	Jena			Blazegraph			RDF-3X			RDFLib			Allegrograph			Virtuoso			HodDB		
	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}	μ	σ^2	99^{th}
`bf:feeds`	16	9	41	29	19	58	10	3	17	62	17	98	29	13	45	24	7	40	8	3	14
`bf:feeds+`	19	14	39	61	20	93	-	-	-	140	20	201	89	21	137	40	12	64	11	4	23
`bf:feeds*`	20	11	51	63	21	105	-	-	-	141	21	200	92	21	140	40	11	66	11	4	22

Table 7.4: Effect of property path operators on query execution time. All times are in milliseconds. This microbenchmark was run against the Soda Brick model.

additional edge during the evaluation of patterns involving `/`, `+` or `*` query operators.

Increased pressure on the "join" mechanism in the executing database tends to be one of the dominating factors in query performance [19, 70]. All databases except for Virtuoso corroborate this effect; the `SensorsInRooms` and `GreyBox` queries consist of over twice as many patterns as the other Brick queries and demonstrate the worst performance of the workload. Virtuoso likely sidesteps this issue because it is built over a relational database with highly optimized joins.

The `*` and `+` property path operators make the query execution time dependent on the depth and size of the matching chains in the graph. Use of these operators effectively increases the number of patterns in the query by the length of the longest predicate chain in the graph, which results in more terms to be joined. To quantify this effect, we run the `AHUChildren` query applying different property path operators to the `bf:feeds` term. Table 7.4 shows the mean, standard deviation and 99th percentile of the resulting query latencies. Allegrograph, Blazegraph, RDFLib and Virtuoso all exhibit a dramatic 200-300% increase in execution time when the query pattern contains the `*` or `+` operators.

This "pattern amplification" happens because `*` and `+` can force a database to resort to slower graph traversal rather than relying on optimized joins between its B-tree indices. The classic YARS-style index [48] used by most SPARQL processors only stores the "next hop" edges and nodes from a given node in the graph. This is a consequence of the YARS index storing each triple individually.

Now that we have established that state-of-the-art RDF databases do not meet the performance target, we need to (1) understand the cause of this deficiency and use this understanding to (2) design a query processor to overcome such performance pitfalls.

7.4 Characterization of Brick Workload

Brick Graph Structure

We first compare several Brick graphs to other RDF datasets commonly used for benchmarking RDF database performance. RDF datasets are commonly characterized by the number of elements (triples, nodes, edges).

	RDF Dataset	Triples	Nodes	Edge Types	Avg Out Degree per Edge Type
Real	Infobox [20]	30,024,092	9,741,482	2063	.0015
	Wordnet [68]	8,574,806	2,487,208	64	.0539
	Sensor [77]	185,950	86,580	12	.179
Syn.	SP2B [91]	7,442	4,800	57	.0272
	BSBM [19]	7,752	3,298	40	.0588
Brick	Soda	8,295	3,429	15	.1613
	SDH	7,458	2,893	13	.1983
	CIEE	359	96	14	.2671

Table 7.5: Graph properties of some published RDF datasets and three representative Brick models.

Table 7.5 compares the size and density of several real-world datasets (DBPedia Infobox [20], LinkedSensor [77] and Wordnet [68]), synthetic datasets (BSBM [19] and SP2B [91]) and Brick models. We draw several conclusions: firstly, Brick graphs are a few orders of magnitude smaller (in number of triples and nodes) and tend not to use as many edge types as other RDF datasets. Secondly, for each edge type, Brick graphs have a higher average fanout. This increases the size of range queries over YARS-style B-tree indices, which can cause a drop in performance.

Impact of SPARQL 1.1 Features on Query Processor

Recall from Chapter 4 that a subset of SPARQL 1.1 features meets the requirements for a Brick query processor (Table 4.2). Table 7.6 enumerates which SPARQL 1.1 features are used by the benchmark SPARQL queries from Figures 7.1 and 7.2 along with the number of triple patterns in each query.

The property path operators `+`, `*`, `?` and `/` allow flexible matching of arbitrary-length chains of relationships. In Soda , the longest chain of `bf:feeds` is of length 2 — from a `brick:AHU` to a `brick:VAV` to a `brick:HVAC_Zone` — so the `AHUChildren` query could be rewritten to explicitly search for `bf:feeds` paths of length 1 and of length 2. However, this would limit the portability of the query and require prior knowledge of the graph structure. The Brick class hierarchy, which has many `rdfs:subClassOf` chains which extend up to a length of 9, exacerbates pattern amplification, especially in queries that use the common `rdf:type/rdfs:subClassOf*` construction.

The implementation of several SPARQL features not required by Brick can affect the performance of a query processor. Most significantly, because the update rate of Brick graphs is low, we can consider a Brick graph to be immutable within a “generation” bookended by batched updates. This removes the need to implement SPARQL `UPDATE`, which adds triples to a graph at any time. Brick also only stores strings — either URIs representing nodes and

Query Name	Patterns	Vars	`+`	`*`	`?`	`|`	`/`	`UNION`
VAV Enum	1	1						
Temp Sensors	1	1		X			X	
AHU Children	2	2	X					
Spatial Mapping	5	3	X					
Room Sensors	11	4	X	X			X	X
VAV Relships	5	5	X					
Grey Box	12	8		X			X	

Table 7.6: Properties of the benchmark SPARQL queries

edges, or literals — and thus does not require implementing numerical constraints or filters.

A Brick query processor should focus on making property path operators performant because these are a primary time consumer, even on small graphs. As we explore in Chapter 8, adopting a batched/generational approach to updating graphs gives a query processor the opportunity to aggressively cache the results of property path operators because apps are likely to query chains of predicates more often than those chains are updated.

Caching the results of a Brick query in an application is discouraged because the application would now operate on stale metadata if the underlying model changes; it is easier to maintain consistency and performance if apps query the model each time and defer this logic to the query processor .

RDF Index Structures

Unsurprisingly, the performance analysis of existing RDF databases suggests that "join" performance is a primary component of SPARQL query execution time. The factors that affect join performance are the time to find the values to join and the time to perform the join itself. Both of these factors depend on the RDF index structure.

Now that we understand the structure of Brick graphs and queries, we delve into how common design decisions made for large-scale RDF graph indices often lack good performance on small graphs with long predicate chains.

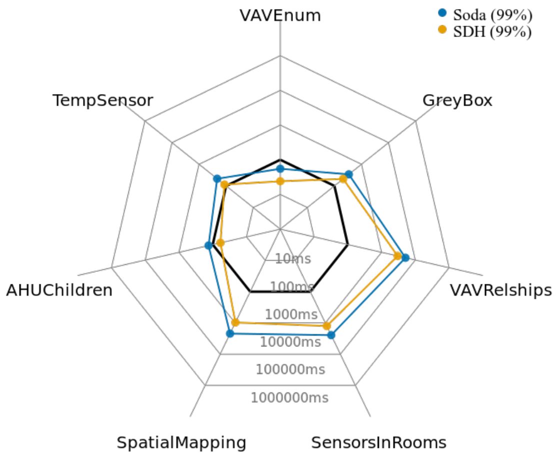
The main reason for this poor performance is the choice of a *triple-oriented* index structure. A triple-oriented index, initially proposed by the YARS query processor [48], uses a collection of B-tree indices to index the dataset by all triples, pairs and single values that could be involved in a query. Each node and edge (**s**ubject, **p**redicate and **o**bject) is assigned a short, unique identifier. Each triple is rewritten using these IDs before being arranged and inserted into six covering indices: SPO, SOP, OSP, OPS, PSO, POS. The indices make use of fast B-tree range traversal to enumerate matching triples; for example, the SPARQL term `?ahu rdf:type brick:AHU` could find all matching subjects by traversing the POS index and looking for all entries with a PO prefix matching the concatenation of `rdf:type` and `brick:AHU`. YARS [48], RDFLib [104], RDF3X [72], Blazegraph [100] and the TDB engine behind Jena [103] all use some form of this index structure.

B-trees are often used as index structures because they have logarithmic scaling properties and provide good spatial locality. However, on small datasets the cost of B-tree range queries can begin to outstrip the rest of the joining computation, and in the case of RDF databases, having multiple separate B-trees is not ideal for maintaining spatial locality. Consider the `AHUChildren` query from Figure 7.1: the query processor will first find all matching subjects for the term `?ahu rdf:type brick:AHU` using the POS index, but cannot reuse that index in order to match the next term `?ahu bf:feeds+ ?x`, which might use the PSO or SPO indices. B-tree spatial locality depends on the order of keys, and because SPARQL queries do not follow lexicographic or numerical orderings, it is difficult to make use of that property. This is especially true the more connected a graph is because there are multiple ways of reaching the same node, which will likely be stored uncontiguously. This effect is exacerbated by the property path operators `/`, `+`, `*` because the sequence of edges to be traversed is only discovered sequentially as the query is evaluated.

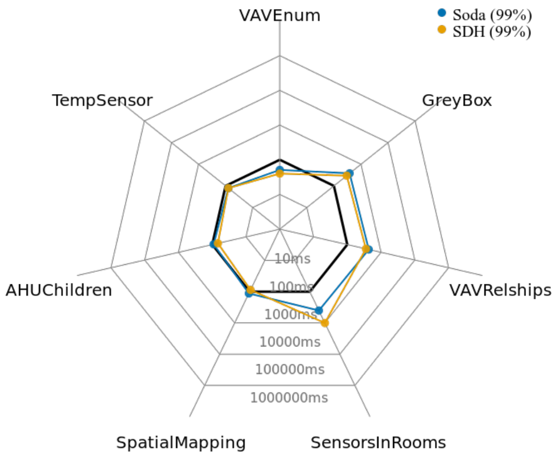
Our findings suggest the typical design decisions made for large sparse RDF datasets do not "scale down" to the small dense graphs typical of Brick. Brick graphs are smaller and tend to have longer predicate chains and a higher out-degree per edge type than other RDF graphs. Further, in contrast to many RDF workloads Brick queries are written to traverse a family of graphs, rather than a specific instance. As a result, Brick queries use many SPARQL 1.1 operators — `UNION` or the `+`, `*` and `/` property path matching operators — that involve traversing many edges. This use-case presents a challenge for many modern RDF databases which use YARS-style B-tree index structures [48].

7.5 Reflection

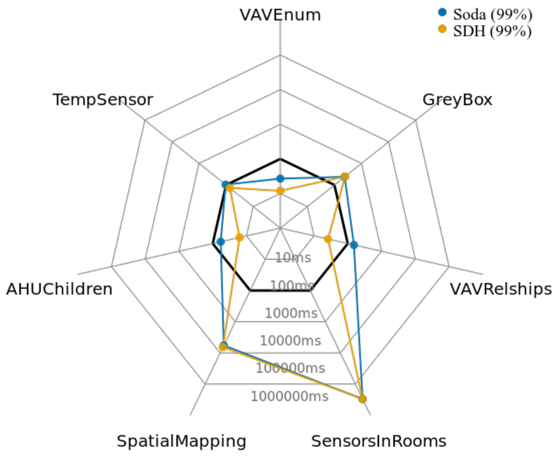
This chapter characterizes the graphs and queries that constitute the latency-sensitive Brick workload and evaluates the performance of existing RDF databases and SPARQL query engines against that workload. The Brick workload consists of Brick graphs that are smaller than other RDF datasets, use fewer edge types (predicates), and possess longer predicate chains. Additionally, Brick queries make heavy use of query operators that match arbitrary-length chains of predicates. Traversing these long chains is intrinsic to the Brick workload because they allow query authors to express uncertainty in the structure of the graph, which increases the portability of queries. These workload properties present performance issues that are pathological to the design of current SPARQL query engines. This motivates the design of a query processor designed specifically for Brick graphs.



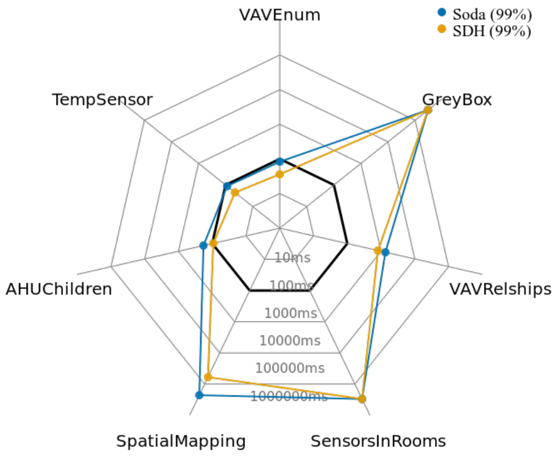
(a) Allegrograph result for Soda and SDH



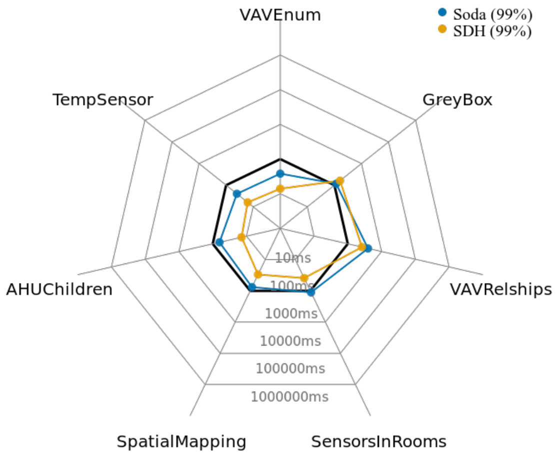
(b) Blazegraph result for Soda and SDH



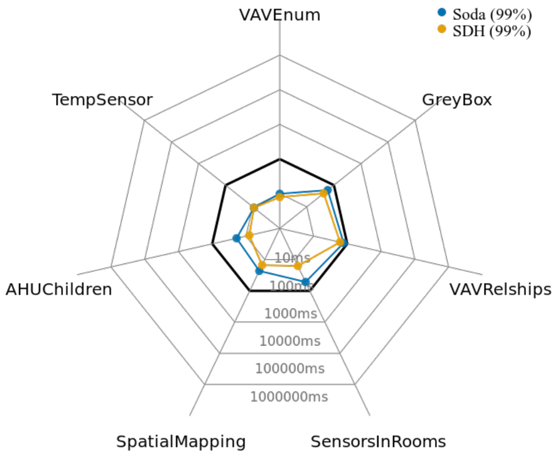
(c) Apache Jena result for Soda and SDH



(d) RDFLib result for Soda and SDH



(e) Virtuoso result for Soda and SDH



(f) HodDB result for Soda and SDH

Figure 7.3: Radar plots showing the 99th percentile latency for each of the benchmark queries over the two larger buildings. All times are in milliseconds. The bold line represents the 100ms target. Note the log scale.

Chapter 8

Design and Implementation of Performant Brick Query Engine

Having established that modern RDF databases do not meet the performance requirements for real-world Brick applications, we now present the design of HodDB, a RDF/SPARQL database specialized for the Brick workload. The key insight is to use the structure of Brick graphs to drive the design of a new RDF index structure that indexes *nodes/entities* rather than full triples. The structure enables a fast graph traversal approach to evaluating SPARQL queries. In addition, the Brick workload enables several simplifying assumptions that can increase performance: (1) take advantage of a read-heavy workload with rare, batched writes to implement aggressive caching, (2) cache inferences by saving chains of predicates as they are traversed, and (3) implement the subset of SPARQL 1.1 features that fulfill the requirements of a Brick query language (Chapter 4).

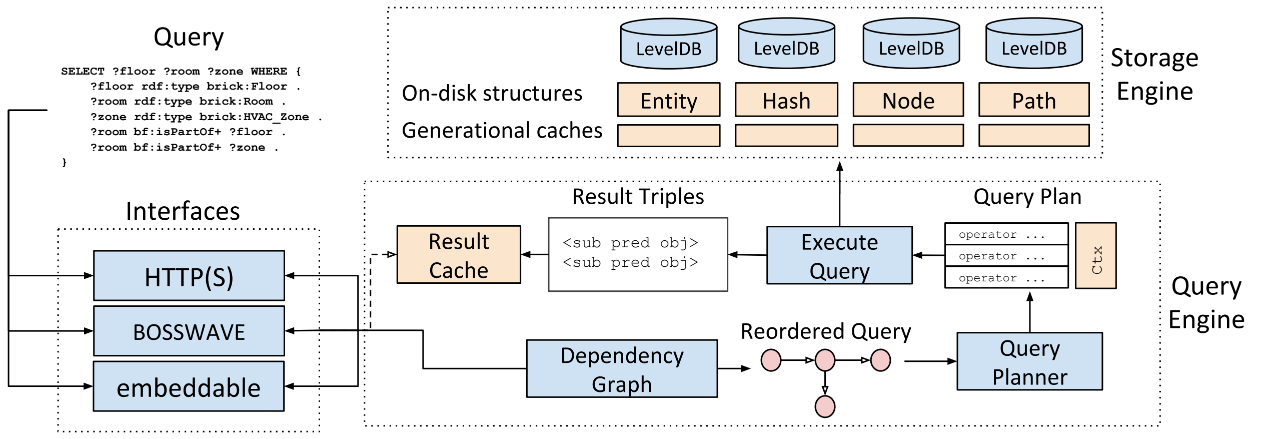
We first present an architectural overview of the HodDB storage engine and index, and then discuss how the HodDB query engine uses the index to evaluate SPARQL queries, followed by an evaluation of HodDB on the established Brick workload. The discussion below refers to the architectural overview in Figure 8.1.

We built HodDB mostly as an exploration of why other RDF databases were so slow on the Brick workload. As a result, HodDB follows standard design paradigms and has not been subjected to a concentrated optimization effort, but nonetheless presents an interesting alternative design point in the RDF database space.

8.1 Design of Multi-Index RDF Storage Engine

HodDB stores the RDF triples constituting a Brick graph in a family of index structures, each implemented over LevelDB [1], a popular embedded key-value database with support for range queries and transactions. All HodDB indices are built over a key-value abstraction.

[1]We use a Go port: `https://github.com/syndtr/goleveldb`



```
SELECT ?floor ?room ?zone WHERE {
    ?floor rdf:type brick:Floor .
    ?room rdf:type brick:Room .
    ?zone rdf:type brick:HVAC_Zone .
    ?room bf:isPartOf+ ?floor .
    ?room bf:isPartOf+ ?zone .
}
```

Figure 8.1: Architecture of HodDB

Entity and Hash Index: RDF triples consist of URIs and literal values, which tend to be large. On the SDH dataset, the average triple uses 174 bytes with the full URIs, and 50 bytes without. As a result, most RDF databases do not work directly with the raw URIs and literals. Instead, many databases use a dictionary to translate between long strings and short unique numerical identifiers; for example, Blazegraph assigns each URI a unique 8-byte integer value and Jena uses a 16-byte MD5 hash.

HodDB uses a 4-byte hash of the string value, calculated using the Murmur3 hash function which has been shown to have good performance and minimal hash collisions. While nothing architecturally prevents HodDB from using larger hashes and supporting more than 2^{32} entities in a graph, we do not believe Brick graphs will ever reach this size, and using 4-byte values instead of 8 or 16-byte values decreases the index size and thus reduces byte movement.

HodDB saves a 2-way mapping between a string and its 4-byte hash. The Entity Index (Figure 8.1) stores the mapping from string to 4-byte hash, and the Hash Index stores the inverse. The rest of the storage and query engines operate entirely on these hashes, which are translated back into the original string values only when the query results are returned. Hash conflicts can arise when new entities inserted; HodDB handles hash conflicts by appending nonces to the input to the Murmur3 hash function and storing the first nonce that successfully yields a new hash. This cost is only incurred the first time HodDB sees an entity; subsequent inserts of the same entity are idempotent.

Node Index: The node index stores a fully elaborated adjacency list representation of the RDF graph. The index keys are the 4-byte hashes of all subject and object entities in the graph; no distinction is made between whether an entity was used as a subject or object in the key. Each index value contains 2 MsgPack [41]-encoded dictionaries: `In` and `Out`. `In` associates the 4-byte hash of a predicate with an array of subject 4-byte hashes for which the keying entity was the object. `Out` does the same but for RDF triples in which the keying entity was the subject.

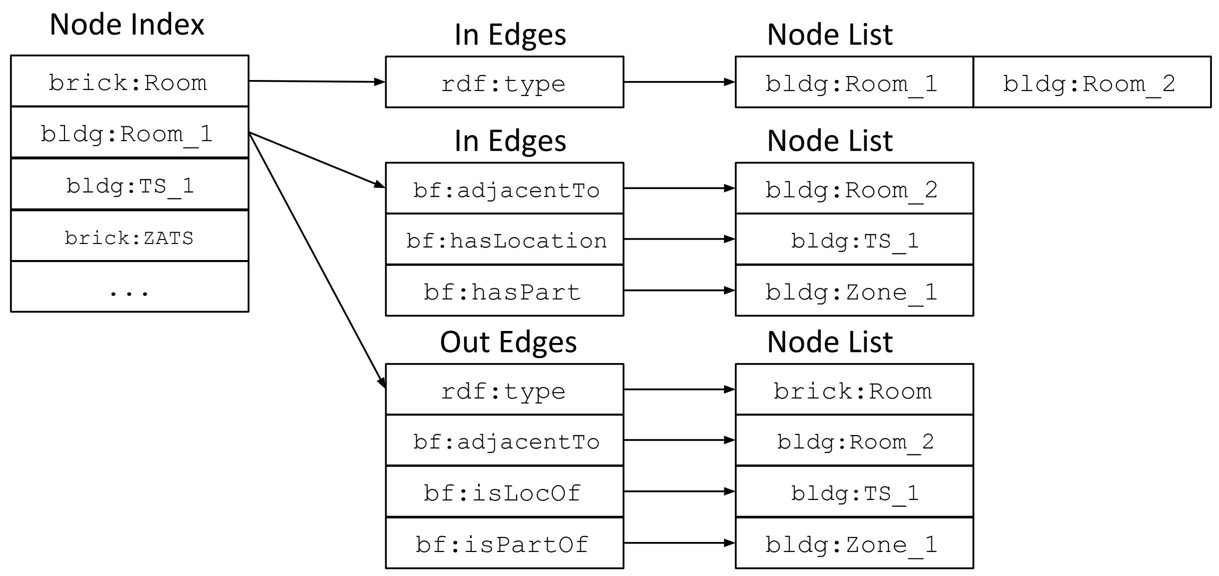


Figure 8.2: HodDB Node Index structure for a small graph.

Figure 8.2 shows this structure for the `brick:Room` and `bldg:Room_1` entities in Figure 8.3. Because Brick defines inverses for the `bf:adjacentTo`, `bf:feeds`, `bf:isLocationOf` and `bf:isPartOf` edges in the original graph, the node index populates the inverse edges in the index even though the triples were not explicitly defined in the source. This obviates the need for the elaborate preprocessing step we applied to other RDF databases that inserts inverse relationships into the graph (Chapter 7).

There are several benefits to this structure. The first is because the index is keyed by individual entities: the query engine only needs one `get()` operation against the backing key-value store to get all triples involving that entity as either a subject *or* an object. This gives good spatial locality; many Brick queries tend to access several edges for the same entity, so having the set of in- and out-edges already in memory while continuing to evaluate a query avoids unnecessary trips to the backing key-value store..

Secondly, this structure accelerates the process of finding candidate values to join during query evaluation. We can decompose the performance of a join into two components: assembling the two sets to be joined, and performing the join itself. The join mechanism is described in §8.2. In denser graphs that have a higher average fanout per node, like Brick models (Figure 7.5), iterating through a B-tree index can result in worse performance than HodDB's approach of simply serializing the list of edges. This is one possible explanation for why HodDB has better performance on the `VAVEnum` query, whose performance depends most directly on this property (Tables 7.1, 7.2 and 7.3). The **predicate index** is similar in structure to the node index, but uses predicate/edge hashes as keys.

Path Index: The path index accelerates evaluation of queries involving chains of

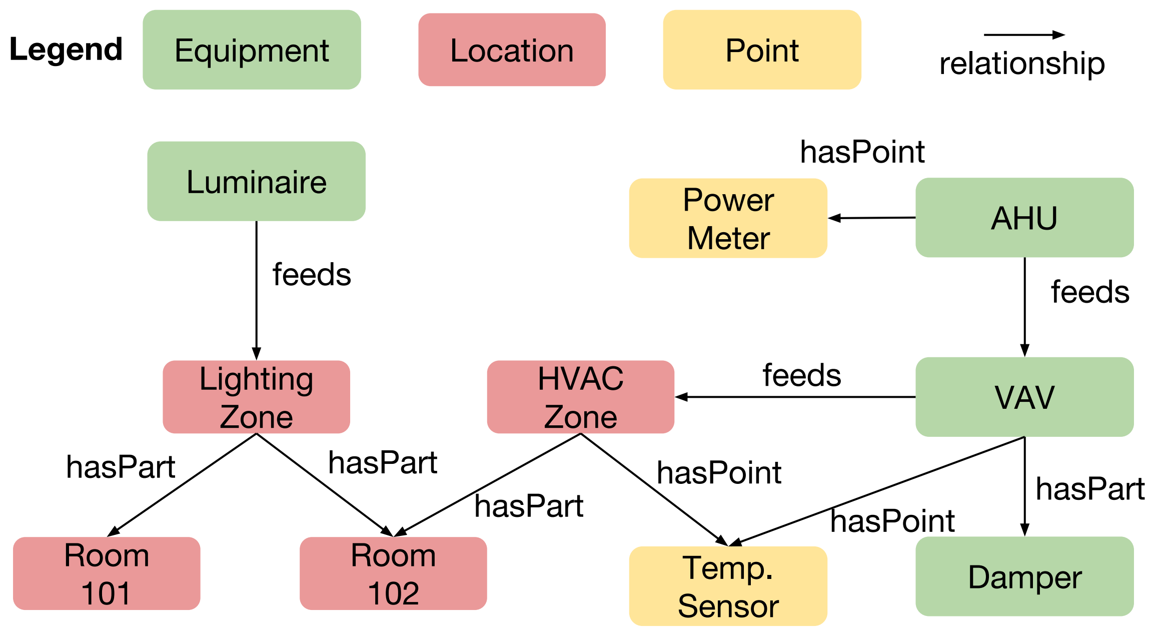


Figure 8.3: Brick classes and relationships (constituting a Brick model) for the HVAC and lighting processes in the sample building in Figure 3.3.

predicates by caching the set of connected entities the first time the query is run. When HodDB sees a query pattern involving `+` or `*`, it checks the path index using the 4-byte hash of the chained predicate, e.g. `rdfs:subClassOf*`. If the entry does not exist, HodDB evaluates the query using graph traversal (as explained below), and keeps track of all entities matched when evaluating the chained predicate. It saves the result in the path index, which has the same structure as the node index, but stores full set of "1 hop or more" entities in the `In` and `Out` dictionaries. For all subsequent queries involving that chain, HodDB can use the cached results.

Like most other caches in HodDB, the path index is discarded when new data is loaded in. Data ingestion is rare enough in current Brick workloads that the cost of rebuilding the path index is not prohibitive, thanks to HodDB's fast graph traversal. Future releases of HodDB will use background processing to preemptively rebuild the path index when this happens.

8.2 Design of SPARQL Query Engine

We now describe HodDB's query evaluation engine, depicted in Figure 8.1. HodDB adopts a graph-traversal approach to evaluating SPARQL queries: starting from an initial set of entities, HodDB uses the patterns in a SPARQL query to direct a traversal of the graph using the node and path indices. We now follow the sequence of steps involved in evaluating

SPARQL Pattern	Operation
`?s p o`	Resolves all nodes `?s` that have edge `p` to node `o`.
`s ?p o`	Resolves all nodes `?p` that connect nodes `s` and `o`.
`s p ?o`	Resolves all nodes `?o` to which node `s` has edge `p`.
`?s ?p o`	Resolves all pairs of nodes `?s` and edges `?p` that connect to `o`.
`s ?p ?o`	Resolves all pairs of edges `?p` and nodes `?o` that originate at `s`.
`?s p ?o`	Resolves all pairs of nodes `?s`, `?o` connected by edge `p`
`?s ?p ?o`	Resolves all triples.

Table 8.1: List of all HodDB query operators. Each operator executes against resolved values in the query context when possible to avoid having to query the node index.

a query in HodDB.

Dependency Graph: HodDB parses SPARQL queries into a set of patterns qualified by the *number* and *name* of the variables they contain. Most patterns look like RDF triples but with one or more of the `subject`, `predicate` and `object` term replaced with a variable (e.g. `?vav rdf:type brick:VAV`). HodDB arranges the patterns into a DAG representing the dependencies between them: a pattern A is dependent on a pattern B if B is more restrictive (contains fewer variable terms) than A and B contains at least one variable from A.

Query evaluation starts at the sink nodes of the dependency DAG, which are the most restrictive patterns. More restrictive patterns allow the query evaluator to "resolve" a variable to a set of candidate entities, which can then be carried through the set of patterns and joined with other sets to build up the result set. An important property of the dependency graph is that it decouples the expression of a query from its execution; in many RDF databases, the order of triple patterns can severely impact execution time [98]. HodDB's dependency graph serves as a basic form of selectivity estimation for reducing the number of entities that need to be joined because more restrictive patterns tend to resolve to fewer candidate entities.

Query Planner: The query planner serializes the dependency graph into a flat list of triple patterns and associates an operator with each pattern according to the positioning of variables in that pattern (Table 8.1). An operator is a small piece of code that takes a triple pattern and a query context as arguments and, using the node and path indices, performs the requisite graph traversals and joins to further filter or expand the set of candidate result entities.

Query Executor: The query executor runs the list of operators output from the query planner, using a query context object to store all intermediate state. Once all operators have been executed, HodDB iterates through the rows in the query context relation and extracts the values corresponding to the variables in the `SELECT` clause. Up until this point, HodDB operates entirely on the 4-byte hashes of the entities; when generating the result set, HodDB uses the hash index to translate the hashes into the actual string values.

Query Operators: Each query operator contains a relation keyed by the variables in its corresponding query pattern. For each query, HodDB creates a query context containing a relation keyed by all variables in the query. When each operator executes, if a variable contained in the operator has values in the context relation, then the operator uses those values instead of referring to the node index. This aids performance because the values in the context relations are in memory and typically smaller than the full graph. An operator with two or more variables may consult both the query context relation and the node index to find the necessary values.

During query execution, HodDB constructs a bitmap for each binding of a variable to a value encoding which rows of the context relation contain that value. HodDB joins the output of each operator with the context relation to form the results of the query using a modified bitmap join [26]. HodDB uses the Go port of the Roaring Bitmap library [25].

HodDB's relation objects contain an index mapping each value of each variable to a bitmap of the rows where that variable has been resolved to that value. Joining two relation objects (always between an operator's relation and the query context relation) involves performing a logical `AND` between the two relation objects for each value of each variable used as the target of the join. The output of the `AND` yields a set of rows whose values are copied into the query context relation. These relations are created anew for each query: the number of possible indexes for a triple pattern is too large for HodDB to create them all at insert time.

Result Cache: One benefit of the batched update model is HodDB knows it only needs to evict its caches when a new update arrives. Between updates, HodDB can optionally cache query results to avoid reevaluating a query when the underlying Brick model has not changed. The HodDB result cache is keyed by an pattern-order-agnostic representation of SPARQL queries, so queries do not have to be byte-equivalent in order to hit the result cache. [2] We disabled the result cache for all measurements of HodDB, but it generally returns results in <1ms on a cache hit.

8.3 Performance Evaluation of SPARQL Query Engine

Microbenchmarks: Referring back to Table 7.4, HodDB's path index means that property path operators only induce a 38% overhead on query execution time. Table 8.2 compares disk usage for each graph for each database. HodDB does not apply specialized compression techniques; nonetheless, these results indicate that HodDB's optimized index structure does not raise any disk utilization concerns.

HodDB's design decisions target small, dense graphs that typify Brick models. Buildings, and therefore Brick models, will only get so large, but an obvious question is how well HodDB

[2] Which is how MySQL's optional result cache works

Building	Jena	Allegrograph	Blazegraph	RDFLib	Virtuoso	RDF-3X	Hod
CIEE	1.5MB	522MB	4.9MB	7.2MB	47MB	800KB	668KB
Soda	5.5MB	522MB	8.8MB	16MB	47MB	2.1MB	2.0MB
SDH	2.5MB	522MB	6.0MB	9.6MB	47MB	1.2MB	1.6MB

Table 8.2: Disk space usage for each graph. HodDB's indices are small — about the same size as RDF-3X's compressed B-trees.

Number of VAVs	1	10	100	1000	10,000,	50,000	100,000
Execution time of `VAVEnum`	.58	.60	1.12	5.28	52.89*/20.02	354.63*/121.82	861.62*/309.52

Table 8.3: Microbenchmark to estimate the impact of Brick model size on HodDB performance. `VAVEnum` query against 6 progressively larger Brick models consisting entirely of VAV instances, constituting a "worst-case" scenario. We observe that HodDB can maintain sub-100ms query latencies for graph sizes <50k triples. The starred values are the full execution time, dominated by the time for the benchmarking client to handle the amount of data being returned. The unstarred values are the raw query execution time of the database, ignoring any time spent on the client.

scales to larger graphs, and at what point do the design trade-offs swing in favor of the common YARS-style triple-oriented indices used by most RDF databases.

To estimate the scaling properties of HodDB, we construct a "worst-case" Brick model consisting of N instances of Variable Air Volume (VAV) boxes and measure the query latency of the `VAVEnum` query from our earlier evaluation (Figure 7.1). This scenario constitutes one of the worst-case scenarios for HodDB's node index. Because the node index is fully elaborated and all nodes in the graph are 1-hop away from each other (all connected through the `brick:VAV` class node), each of the N nodes needs to store the other $N - 1$ nodes in its entry. This increases both the read time from the underlying LevelDB store, but also the deserialization time for the index entry on the first load. As such, we would expect the execution time for the `VAVEnum` query to be exponential in the number of VAV instances.

Table 8.3 contains the result of this experiment for exponentially large numbers of nodes. Ignoring client latency, we find that HodDB starts to miss the sub-100ms target query latency around the size of a graph of 50,000 VAV instances. For context, the two large buildings (each with less than 10,000 nodes) used in the §7.2 evaluation are representative of Brick model size and complexity. The benchmark results suggest that the current design of HodDB will be sufficient for existing use cases of Brick, but significant slowdown is possible in extreme cases. Future evaluation should adopt methods from [19] or [91] for autogenerating RDF graphs that follow an approximate structure; this would yield a more realistic graph to use for scalability measurements.

Brick Workload: We now refer back to Tables 7.1, 7.2 and 7.3; the last column shows the query latency distribution for HodDB. The mean latencies are all below 50ms, and the 99th percentile latencies (influenced mostly by garbage collection pauses) are all below the performance target of 100ms. To more closely emulate a real deployment these query results all include the overhead of the benchmarking Python client, which contributes a small <4ms latency to all requests.

8.4 Metadata-driven Applications Enabled by HodDB

HodDB enables new modes of interaction with a Brick model. We explore three applications here: The first application is an interactive query interface that progressively visualizes the class structure of a Brick graph in response to user input. The second application uses Brick queries to define the structure of data matrices such as for training models or performing an analysis. The third application is a scheduler service that uses Brick queries to define control relationships.

Interactive Query Visualizer

Visualization is a common technique for making sense of RDF graphs [89, 39, 49, 32]. Most approaches either have users start from a node in the graph and explore outwards, or start from the full graph and apply filters to restrict what is shown. The problem with these approaches is they either limit the generalizability of the visualization (starting from a single node and exploring outward does not inform the user about the larger structure of the graph), or requires the user to be familiar with the structure of the graph (such as to write effective filters to restrict the graph).

Another approach used by tools such as Protege [74] is to visualize the ontologies used in an RDF graph. The corresponding visualizations are often much smaller and more manageable than the full graph. However, they only inform the user about the general structure of graphs using that ontology, rather than the structure of a specific graph. The Brick ontology contains hundreds of classes of equipment and points that do not exist in every building. A naive visualization of the Brick ontology would not inform a user which classes are used in a particular Brick model, and how those classes are related.

HodDB proposes a distinct method for Brick models that allows a user to progressively explore a Brick model's *class structure*. The key idea for class structure visualization is that the cardinality and complexity of a graph visualization can be mitigated by showing how *types* of nodes are connected, rather than how the nodes themselves are connected. All nodes in a Brick model are instances of one or more classes, so HodDB can extract the class structure of a graph by clustering nodes by their class. Each node's class can be found by following its `rdf:type` edge; every node in a Brick model has an `rdf:type` edge. HodDB can perform this for a full Brick model as well as the results of a SPARQL query against that model, which enables progressive visualization of the class structure.

HodDB v0.5.5 [3] and onward ship with a web frontend implementation of this method. Figure 8.4 is a screenshot of a sample interaction. The user begins with the

[3] `https://github.com/gtfierro/hod/releases/tag/v0.5.5`

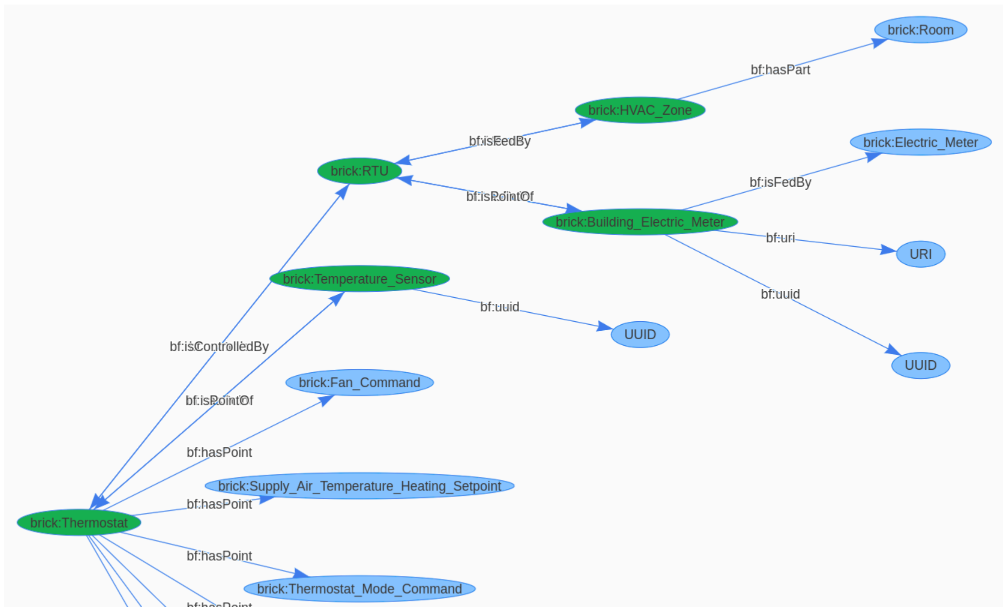


Figure 8.4: Demonstration of class structure visualization in HodDB. Highlighted nodes are those selected for expansion by the user.

`brick:Thermostat` class node. Clicking that node reveals the relationships (edges) and classes of the "1-hop" neighbors of every instance of `brick:Thermostat` in the model. Of these newly revealed classes, the user selected the `brick:RTU` (Rooftop Unit) class, then the `brick:Building_Electric_Meter` and `brick:HVAC_Zone` classes. Users can also deselect nodes to collapse their edges.

HodDB tracks each node the user clicks and generates a valid SPARQL query corresponding to the revealed class structure that resolves to the actual instances of those classes. Figure 8.5 contains an example of a generated query. HodDB resolves these generated queries and then applies the class structure visualization method described above. Each node the user clicks adds three more patterns to the generated SPARQL query, requiring a large number of joins during query execution. HodDB's procedure for generating these three patterns is as follows:

- Generate a new SPARQL variable $?X$
- Output a pattern linking this variable $?X$ to the existing query; this involves retrieving the label of any edge connecting the clicked class node to the rest of the graph
- Output the triple pattern identifying all nodes $?X$ that are instances of the class clicked by the user: `?X rdf:type <clicked class> .`
- Generate 2 new SPARQL variables $?P$ and $?O$

```
SELECT ?7ee7 ?4388 ?3dc8 ?1e9b WHERE {
    ?7ee7 rdf:type brick:Thermostat .
    ?7337 bf:controls ?4388 .
    ?4388 rdf:type brick:RTU .
    ?4388 bf:hasPoint ?3dc8 .
    ?3dc8 rdf:type brick:Building_Electric_Meter .
    ?4388 bf:feeds ?1e9b .
    ?1e9b rdf:type brick:HVAC_Zone
}
```

Figure 8.5: Autogenerated query from the interaction in Figure 8.4. Variable names are autogenerated. The `SELECT` clause contains the variables representing nodes selected by the user. HodDB drops the autogenerated terms containing 3 variables when returning the query to the user

- Output the triple pattern identifying all outgoing edges and nodes for the instances ?X: `?X ?P ?O .`. This is dropped when exposing the query to the user (Figure 8.5).

Optimization of the generated queries is an area of future work. Currently, generated queries do not make use of the `?`, `+`, `/` or `*` operators. Fortunately, the design of HodDB's query processor means that the increasing complexity of the query does not hinder the responsiveness of the interface.

These generated queries can also be returned to the user, which means that a user can simultaneously and instantaneously view both the textual representation of a query and a digestable graphical representation of that query's results. We have found this to be a powerful tool in introducing new users to the Brick schema.

Integrating Brick with HodDB

HodDB enables building applications to take advantage of the contents and structure of a Brick model. Applications can embed application-specific metadata in a Brick model by adding RDF literals (strings) as nodes in the graph and relating these to existing nodes in the model. We will explore two services that use HodDB to tightly integrate with a Brick model.

Brick-driven Datasets

The Metadata-driven Data Access Layer (MDAL) service binds points of sensing and actuation as represented in a Brick model to streams of historical data stored in a timeseries database. Users use Brick queries to describe the building data they want to download from MDAL, meaning that the retrieval of datasets can be as portable as a Brick query. The specific datasets retrieved through MDAL will be different from building to building, but their structure will be consistent which aids in the construction of portable analytics and model training.

```
@prefix bf: <https://brickschema.org/schema/1.0.1/BrickFrame#> .
@prefix brick: <https://brickschema.org/schema/1.0.1/Brick#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix bldg: <http://brickuniversity.edu/buildings/BuildingABC#> .
bldg:Temp_Sensor_1 rdf:type brick:Zone_Air_Temperature_Sensor .
bldg:Temp_Sensor_1 bf:uuid "1a5a7224-fa34-11e7-823a-1002b58053c7" .
```

Figure 8.6: Example of augmenting a Brick model representation of a temperature sensor with a 36-byte UUID used for historical data access.

MDAL makes use of existing monitoring infrastructure [4] that periodically polls the state of all points in a building (such as temperature sensors and damper position setpoints) and persists this data in BTrDB [5], a fast and scalable timeseries database. BTrDB uniquely identifies each point (sensor, actuator, setpoint) using a 36-byte UUID, which can be used to retrieve the historical values of that point. MDAL adds these 36-byte UUIDs to a Brick model as RDF literals, and relates them to the correct point using the edge label `bf:uuid` (Figure 8.6).

MDAL queries describe the composition of the desired dataset; Figure 8.7 contains a representative query. The `Variables` and `Composition` parameters define which points constitute the desired dataset. The `Variables` parameter defines the collections of points in the Brick model desired by the user. If a Brick model includes the engineering units of a point, then MDAL can also provide unit conversion. In Figure 8.7, the user is interested in both inside and outside temperature. When MDAL executes a query, it evaluates the Brick queries in each variable definition to the corresponding set of UUIDs. These are substituted into the `Composition` parameter, which defines the columns of the returned dataset. The `Selectors` and `Time` parameters are passed through to BTrDB to specify the resampling policy and temporal range of the desired dataset.

Brick-driven Scheduler

A scheduler is a simple example of how HodDB can store configuration for a control process in a building. As with MDAL, the scheduler uses RDF literals to embed external configuration inside a Brick model. Our simple scheduler augments a Brick model with REST API endpoints of the thermostats in a building: the endpoint URLs are encoded as RDF literals and associated with thermostat nodes using a new `bf:uri` relationship.

The simplest scheduler we can write using this augmentation of the Brick model can discover all controllable thermostats in a building and modulate them on some shared schedule. Figure 8.8 contains Python pseudocode for such a schedule. Every minute, the scheduler executes a Brick query that retrieves all of the controllable thermostats in the building along with their API endpoints and which HVAC zone they control. The advantage of evaluating

[4]Provided by the XBOS Project: `https://docs.xbos.io/`

```
{
  "Composition": ["internaltemp", "externaltemp"],
  "Selectors": ["mean", "mean"],
  "Variables": [
    {
      "Name": "internaltemp",
      "Definition": """SELECT ?uuid WHERE {
                          ?sensor rdf:type/rdfs:subClassOf* brick:Temperature_Sensor .
                          ?sensor bf:uuid ?uuid .
                          ?sensor bf:hasLocation ?r .
                          ?r rdf:type brick:Room
                      };""",
      "Units": "C",
    },
    {
      "Name": "externaltemp",
      "Definition": """SELECT ?uuid WHERE {
                          ?sensor rdf:type/rdfs:subClassOf* brick:Outside_Temperature_Sensor .
                          ?sensor bf:uuid ?uuid
                      };""",
      "Units": "C",
    },
  ],
  "Time": {
    "T0": "2017-08-01 00:00:00",
    "T1": "2017-08-31 00:00:00",
    "WindowSize": "1h",
    "Aligned": "True",
  }
}
```

Figure 8.7: Sample MDAL query retrieving internal and external temperature data for a deployment resampled to 1-hour means.

this query every time the scheduler executes is the scheduler will automatically discover and actuate new or altered thermostats without any administrative intervention. In this way, the Brick model acts as the “single point of truth” for the current configuration of a building. Writing controllers to consult the Brick model means they will always have a consistent view.

We can combine MDAL with a scheduler service to implement a portable thermostat controller that learns occupancy schedules. First, the controller executes a simple Brick query to retrieve all of the HVAC zones in the building (`?zone rdf:type brick:HVAC_Zone`). For each zone, the controller then constructs a query that retrieves the API endpoint for that zone’s thermostat and UUIDs of all occupancy sensors for all rooms in that HVAC zone. The controller uses this query in an MDAL request to fetch recent occupancy data for that zone and trains a model to predict an occupancy schedule for that zone. The query also yields the API endpoint needed to enact the schedule according to the occupancy predictions. Figure 8.9 contains the Python pseudocode for part of this controller. For simplicity, we elide the bookkeeping code required to decouple the training of these occupancy models from their execution as schedules.

```
while True:
  thermostats = hoddb.run_query("""
  SELECT ?tstat ?api ?zone WHERE {
      ?tstat rdf:type brick:Thermostat .
      ?tstat bf:uri ?api .
      ?tstat bf:controls/bf:feeds+ ?zone .
      ?zone rdf:type brick:HVAC_Zone
  }
  """)
  for thermostat in thermostats:
    print "Scheduling thermostat {0} for zone {1}".format(thermostat["?tstat"], thermostat["?zone"])
    if time.after("18:00"): # 6pm, start nighttime schedule
      POST(thermostat["?api"], {"heating_setpoint": 60, "cooling_setpoint": 85})
    if time.after("8:00"): # 8am, start daytime schedule
      POST(thermostat["?api"], {"heating_setpoint": 72, "cooling_setpoint": 76})
  time.sleep(60) # run every minute
```

Figure 8.8: Python pseudocode for a simple thermostat controller that enacts the same schedule over all HVAC zones

```
zones = hoddb.run_query("SELECT ?zone WHERE { ?zone rdf:type brick:HVAC_Zone }")
for zone in zones:
  query = """SELECT ?uuid ?room ?tstat_api WHERE {
    ?room bf:isPartOf {0} .
    ?room bf:isLocationOf ?sensor .
    ?sensor rdf:type/rdfs:subClassOf* brick:Occupancy_Sensor .
    ?sensor bf:uuid ?uuid .
    {0} bf:isFedBy+/bf:isControlledBy ?tstat .
    ?tstat rdf:type brick:Thermostat .
    ?tstat bf:uri ?tstat_api
  }""".format(zone["?zone"])
  tstat_config = hoddb.run_query(query)
  # construct MDAL query with the following variable definition
  mdalquery["Variables"] = [{
      "Name": "occ",
      "Definition": query,
  }]
  occupancy_data = mdal.run_query(mdalquery)
  model = train_model(occupancy_data)
  schedule = schedule_from_model(model)
  execute_schedule(tstat_config["?tstat_api"], schedule)
```

Figure 8.9: Python pseudocode for a thermostat controller that learns its schedule from a model trained on occupancy data from the rooms conditioned by the thermostat

These services illustrate how HodDB can serve as a point of integration between a Brick model – a logical representation of the resources in a building and how they are related – and the infrastructure that performs the monitoring and control of those resources. Constructing services, controllers and analytics to retrieve their configuration information through a portable Brick query means these processes can be deployed on multiple buildings without an intensive manual effort.

8.5 Reflection

We use the characterization of the Brick workload to develop HodDB, a new RDF/SPARQL query processor built around an alternative RDF index structure providing fast query evaluation. HodDB consistently meets the 99th percentile latency target of 100ms, and enables a new class of portable, metadata-driven, Brick-based applications for advanced control and monitoring of heterogeneous buildings. Finally, we demonstrate several new applications enabled by HoddB's quick execution of Brick queries. The developed applications push the state of the art in how RDF models are visualized and integrated with analytics and control services in the built environment.

Chapter 9

Conclusion and Future Work

9.1 Industrial Collaboration

The American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE) is a professional society and body for standards development. The BACnet committee within ASHRAE is developing standard 223P – “Designation and Classification of Semantic Tags for Building Data” – which aims to enable semantic interoperability in buildings. The teams behind Brick and Project Haystack are collaborating with the BACnet committee on the development of the standard[1]. Effort is ongoing, but a public release of the standard is expected in 2019.

Project Haystack has formed a new working group (WG551)[2] to create a type system for tags applied to entities. The goals are to reduce ambiguity in combinations of tags (discussed in this thesis) and to automatically generate documentation for valid combinations of tags. The type system will be formalized using an ontology expressed in the RDF data model, which will enable it to be linked directly with the Brick schema.

Johnson Controls, a large international BMS vendor, has invested in the creation of Brick Consortium, Inc, a nonprofit organization focused on developing the Brick schema and specifications for Brick-compliant products.

9.2 Brick Developments

Brick is central to several developing projects.

Mortar[37] is an open testbed for the development and evaluation of building analytics software. It contains timeseries data for over 100 buildings, spanning 10 billion data points

[1]`https://web.archive.org/web/20181223045430/https://www.ashrae.org/about/news/2018/ashrae-s-bacnet-committee-project-haystack-and-brick-schema-collaborating-to-provide-unified-data-semantic-modeling-solution`

[2]`https://web.archive.org/web/20181120082921/https://project-haystack.org/forum/topic/551`

and 26,000 data streams. Each of the buildings has an associated Brick model that provides the association between points in the building subsystem and their corresponding archival data streams. Mortar aims to create an open library of data analytics applications that leverage Brick to make their implementations portable across buildings.

XBOS is an open-source distributed operating system for buildings that can interface with a wide array of building technologies from building management systems to off the shelf commercial devices such as smart thermostats. XBOS enables real-time monitoring and control of building systems, the collection, modeling and analytics of building data, and advanced management and coordination of building systems. XBOS uses Brick to describe the building systems it interfaces with as well as the generic interfaces provided by XBOS. Software that interacts with XBOS can thus remain agnostic to the particular APIs for the underlying BMS and other hardware contained in a building. This enables portable building management software.

9.3 Conclusion

The heterogeneity of building representation presents a major bottleneck to the fast and low cost deployment of energy efficiency applications. This thesis presents the design and implementation of Brick, a new standard metadata schema for buildings that fulfills the need for an expressive, complete and usable description of buildings for enabling portable applications. Portable applications offer a means for the broad deployment of energy efficiency applications.

Brick is motivated by the failure of existing metadata standards to capture the entities and relationships required by the suite of existing and future energy efficiency applications. Existing standards provide brittle, application- or vendor-specific descriptions of existing building subsystems, but are unable to capture the structure or processes of those systems in a general way. Brick is defined with an ontology expressed in the RDF data model. The ontology defines Brick's extensible class hierarchy, which organizes the valid types of building entities and assets, and Brick's relationships, which express the context necessary for portable applications. The design of Brick is shown to describe more than 98% of points and equipment defined in existing BMS – demonstrated on six real-world buildings – and generalizes gracefully to diverse building subsystems.

The standard SPARQL query language is shown to fulfill the requirements of a Brick query language, and is able to express the application requirements demonstrated by 15 real-world applications. However, most existing implementations of SPARQL are not sufficiently performant to meet the latency requirements of Brick queries used in real applications.

The thesis characterizes the graphs and queries that constitute the Brick workload, finding that Brick graphs are smaller than other RDF datasets, use fewer edge types (predicates), and possess longer predicate chains. Brick queries make heavy use of query operators that match arbitrary-length chains of predicates. Traversing these long chains is intrinsic to the Brick workload because they allow query authors to express uncertainty in the structure of the graph, which increases the portability of queries. The thesis then presents a performance evaluation of current, popular RDF databases against the Brick workload, and demonstrates that none of them meet the latency target of 100ms.

This characterization of the Brick workload is used to to develop HodDB, a new RDF/SPARQL query processor built around an alternative RDF index structure providing fast query evaluation. HodDB consistently meets the 99th percentile latency target of 100ms, and enables a new class of portable, metadata-driven, Brick-based applications for advanced control and monitoring of heterogeneous buildings. This enables several new applications enabled by HoddB's quick execution of Brick queries. The developed applications push the state of the art in how RDF models are visualized and integrated with analytics and control services in the built environment.

Appendix A

Brick Evaluation Queries

```
# find occupancy and luminance sensors in rooms
SELECT ?sensor ?room WHERE {
    ?room rdf:type brick:Room .
    ?sensor bf:hasLocation ?room .
    { ?sensor rdf:type/rdfs:subClassOf* brick:Occupancy_Sensor }
    UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:Luminance_Sensor }
}

# find lighting/HVAC equipment in rooms
SELECT ?equipment ?room WHERE {
    ?room rdf:type brick:Room .
    ?equipment bf:hasLocation ?room .
    { ?equipment rdf:type/rdfs:subClassOf* brick:Lighting_System }
    UNION
    { ?equipment rdf:type/rdfs:subClassOf* brick:HVAC_System }
}
```

Figure A.1: Queries for the Energy Apportionment application

```
# build up spatial breakdown of building
SELECT ?floor ?room ?zone WHERE {
    ?floor rdf:type brick:Floor .
    ?room rdf:type brick:Room .
    ?zone rdf:type brick:HVAC_Zone .

    ?room bf:isPartOf+ ?floor .
    ?room bf:isPartOf+ ?zone .
}

# associate AHUs and VAVs with Zones
SELECT ?vav ?ahu ?hvac_zone WHERE {
    ?vav rdf:type brick:VAV .
    ?ahu rdf:type brick:AHU .
    ?ahu bf:feeds ?vav .
    ?hvac_zone rdf:type brick:HVAC_Zone .
    ?vav  bf:feeds ?hvac_zone .
}
```

Figure A.2: Queries for the Model-Predictive Control application

```
# find all equipment and its control points
SELECT ?equip ?cmd ?status WHERE {
    ?equip  rdf:type/rdfs:subClassOf* brick:Equipment .
    {
        ?cmd rdf:type/rdfs:subClassOf*    brick:Command .
        ?cmd (bf:isPointOf|bf:isPartOf)+ ?equip
    }
        UNION
    {
        ?status rdf:type/rdfs:subClassOf* brick:Status .
        ?status (bf:isPointOf|bf:isPartOf)+ ?equip
    }
}
```

Figure A.3: Queries for the Demand Response application

```
# query to find sensors in rooms / zone
SELECT ?sensor ?room WHERE {
    { ?sensor rdf:type/rdfs:subClassOf* brick:Zone_Temperature_Sensor }
        UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:Discharge_Air_Temperature_Sensor }
        UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:Occupancy_Sensor }
        UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:CO2_Sensor }

    ?vav rdf:type brick:VAV .
    ?zone rdf:type brick:HVAC_Zone .
    ?room rdf:type brick:Room .
    ?vav bf:feeds+ ?zone .
    ?zone bf:hasPart ?room .

    {?sensor bf:isPointOf ?vav }
        UNION
    {?sensor bf:isPointOf ?room }
}

# power meters for equipment in the room
SELECT ?meter ?equipment ?room WHERE {
    ?meter rdf:type/rdfs:subClassOf* brick:Power_Meter .
    ?room rdf:type brick:Room .
    ?equipment rdf:type ?class .
    ?class rdfs:subClassOf+ brick:Equipment .
    ?equipment bf:hasLocation ?room .
    ?meter bf:isPointOf ?equipment
}
```

Figure A.4: Queries for the Occupancy Modeling application

```
# get equipment and power meters
SELECT ?x ?meter WHERE {
    ?meter rdf:type/rdfs:subClassOf* brick:Power_Meter .
    ?meter (bf:isPointOf|bf:isPartOf)+ ?x .
    {?x rdf:type/rdfs:subClassOf* brick:Equipment }
        UNION
    {?x rdf:type/rdfs:subClassOf* brick:Location }
}
```

Figure A.5: Queries for the Non-Intrusive Load Monitoring application

```
# Find reheat/cooling valve commands for VAVs
SELECT ?vlv_cmd ?vav WHERE {
    { ?vlv_cmd rdf:type brick:Reheat_Valve_Command }
        UNION
    { ?vlv_cmd rdf:type brick:Cooling_Valve_Command }
    ?vav rdf:type brick:VAV .
    ?vav bf:hasPoint+ ?vlv_cmd
}

# Airflow sensors for VAVs, map VAVs to HVAC Zones and Rooms
SELECT ?airflow_sensor ?room ?vav WHERE {
    ?airflow_sensor rdf:type/rdfs:subClassOf* brick:Supply_Air_Flow_Sensor .
    ?vav rdf:type brick:VAV .
    ?room rdf:type brick:Room .
    ?zone rdf:type brick:HVAC_Zone .
    ?vav bf:feeds+ ?zone .
    ?room bf:isPartOf ?zone .
    ?airflow_sensor bf:isPointOf ?vav
}

# Find power meters for the HVAC system
SELECT ?equip ?meter WHERE {
    ?meter rdf:type brick:Power_Meter .
    ?meter bf:isPointOf* ?equip .
    ?equip bf:isPartOf* ?thing .
    {?thing rdf:type/rdfs:subClassOf* brick:Water_System }
        UNION
    {?thing rdf:type/rdfs:subClassOf* brick:HVAC_System }
}
```

Figure A.6: Queries for the Web Displays application

```
# associate lighting equipment with rooms
SELECT ?light_equip ?light_state ?light_cmd ?room WHERE {

    ?light_equip rdf:type/rdfs:subClassOf* brick:Lighting_System .

    ?light_equip bf:feeds ?zone .
    ?zone rdf:type brick:Lighting_Zone .
    ?zone bf:contains ?room .
    ?room rdf:type brick:Room .

    ?light_state rdf:type/rdfs:subClassOf* brick:Luminance_Status .
    ?light_cmd rdf:type/rdfs:subClassOf* brick:Luminance_Command .

    ?light_equip bf:hasPoint ?light_state .
    ?light_equip bf:hasPoint ?light_cmd
}

# get room or floor-level power meters
SELECT ?meter ?loc WHERE {
    ?meter rdf:type/rdfs:subClassOf* brick:Power_Meter .
    ?meter  bf:isPointOf ?loc .
    { ?loc    rdf:type    brick:Room }
        UNION
    { ?loc rdf:type brick:Floor }
}
```

Figure A.7: Queries for the Participatory Feedback application

```
# get all sensors for the AHU and for AHU subcomponents
SELECT ?ahu ?sensor WHERE {
    ?ahu rdf:type/rdfs:subClassOf* brick:AHU .
    ?ahu bf:hasPoint|(bf:hasPart/bf:hasPoint) ?sensor .

    { ?sensor rdf:type/rdfs:subClassOf* brick:Reheat_Valve_Command }
        UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:Cooling_Valve_Command }
        UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:Mixed_Air_Temperature_Sensor }
        UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:Outside_Air_Temperature_Sensor }
        UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:Return_Air_Temperature_Sensor }
        UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:Supply_Air_Temperature_Sensor }
        UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:Outside_Air_Humidity_Sensor }
        UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:Return_Air_Temperature_Sensor}
        UNION
    { ?sensor rdf:type/rdfs:subClassOf* brick:Outside_Air_Damper_Position_Sensor }
}
```

Figure A.8: Queries for the Fault Detection and Diagnosis application

Bibliography

[1] Jans Aasman. “Allegro graph: RDF triple database”. In: *Cidade: Oakland Franz Incorporated* (2006).

[2] Yuvraj Agarwal et al. “Duty-cycling buildings aggressively: The next frontier in HVAC control”. In: *10th Information Processing in Sensor Networks (IPSN)*. IEEE. 2011, pp. 246–257.

[3] Yuvraj Agarwal et al. “Occupancy-driven energy management for smart building automation”. In: *Proceedings of the 2nd ACM workshop on embedded sensing systems for energy-efficiency in building*. ACM. 2010, pp. 1–6.

[4] Mohamed H Albadi and Ehab F El-Saadany. “A summary of demand response in electricity markets”. In: *Electric power systems research* 78.11 (2008), pp. 1989–1996.

[5] Michael P Andersen and David E Culler. “BTrDB: optimizing storage system design for timeseries processing”. In: *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)*. 2016.

[6] Renzo Angles and Claudio Gutierrez. “The expressive power of SPARQL”. In: *International Semantic Web Conference*. Springer. 2008, pp. 114–129.

[7] Grigoris Antoniou and Frank Van Harmelen. “Web ontology language: Owl”. In: *Handbook on ontologies*. Springer, 2004, pp. 67–92.

[8] Michael Ashburner et al. “Gene Ontology: tool for the unification of biology”. In: *Nature genetics* 25.1 (2000), pp. 25–29.

[9] Bharathan Balaji et al. “Brick: Towards a unified metadata schema for buildings”. In: *Proceedings of the ACM International Conference on Embedded Systems for Energy-Efficient Built Environments (BuildSys). ACM.* 2016.

[10] Bharathan Balaji et al. “Genie: a longitudinal study comparing physical and software thermostats in office buildings”. In: *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM. 2016, pp. 1200–1211.

[11] Bharathan Balaji et al. “Zodiac: Organizing Large Deployment of Sensors to Create Reusable Applications for Buildings”. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 13–22.

[12] Vladimir Bazjanac and DB Crawley. “Industry foundation classes and interoperable commercial software in support of design of energy-efficient buildings”. In: *Building Simulation’99*. Vol. 2. 1999, pp. 661–667.

[13] J. Beetz, J. Van Leeuwen, and B. De Vries. “IfcOWL: A case of transforming EXPRESS schemas into ontologies”. In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 23.01 (2009), pp. 89–101.

[14] Willy Bernal et al. “MLE+: a tool for integrated design and deployment of energy efficient building controls”. In: *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. ACM. 2012, pp. 123–130.

[15] Tim Berners-Lee, James Hendler, Ora Lassila, et al. “The semantic web”. In: *Scientific american* 284.5 (2001), pp. 28–37.

[16] Arka A Bhattacharya et al. “Automated metadata construction to support portable building applications”. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 3–12.

[17] Arka Bhattacharya, Joern Ploennigs, and David Culler. “Short Paper: Analyzing Metadata Schemas for Buildings: The Good, the Bad, and the Ugly”. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 33–34.

[18] *Bidgely*.

[19] Christian Bizer and Andreas Schultz. *The berlin sparql benchmark*. `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/`. 2009.

[20] Christian Bizer et al. “DBpedia-A crystallization point for the Web of Data”. In: *Web Semantics: science, services and agents on the world wide web* 7.3 (2009), pp. 154–165.

[21] Dario Bonino and Fulvio Corno. “DogOnt – Ontology Modeling for Intelligent Domotic Environments”. In: *ISWC - Int. Semantic Web Conf.* Vol. 5318. 2008, pp. 790–803.

[22] Ken Bruton et al. “Development of an Automated Fault Detection and Diagnosis tool for AHU’s”. In: (2012).

[23] James F Butler and Robert Veelenturf. “Point naming standards”. In: *ASHRAE Journal* 52.11 (2010), B16–B16.

[24] Cayleygraph. *Cayley*. `https://cayley.io`. 2017.

[25] Samy Chambi et al. “Better bitmap performance with Roaring bitmaps”. In: *Software: practice and experience* 46.5 (2016), pp. 709–719.

[26] Chee-Yong Chan and Yannis E Ioannidis. “Bitmap index design and evaluation”. In: *ACM SIGMOD Record*. Vol. 27. 2. ACM. 1998, pp. 355–366.

[27] Victor Charpenay et al. "An ontology design pattern for IoT device tagging systems". In: *5th Int. Conf. on the Internet of Things (IOT)*. IEEE. 2015, pp. 138–145.

[28] Gary E. Choquette. *Tag Naming Conventions and Data Structures for Industrial PLCs*. `http://www.optimizedtechnicalsolutions.com/ReferenceDocuments/Naming%20Conventions%20and%20Data%20Structures%20for%20Industrial%20PLCs.pdf`. 2015.

[29] *Comfy*.

[30] Council on Finance, Insurance and Real Estate. *Financing Small Commercial Building Energy Performance Upgrades: Challenges and Opportunities*. 2016.

[31] LM Daniele, FTH den Hartog, and JBM Roes. *Study on Semantic Assets for Smart Appliances Interoperability: D-S4: FINAL REPORT*. Tech. rep. European Union, 2015.

[32] Leonidas Deligiannidis, Krys J Kochut, and Amit P Sheth. "RDF data exploration and visualization". In: *Proceedings of the ACM first workshop on CyberInfrastructure: information management in eScience*. ACM. 2007, pp. 39–46.

[33] Inc Dgraph Labs. *Dgraph*. `https://dgraph.io/index.html`. 2017.

[34] Energy Star. *Save energy*. `https://www.energystar.gov/buildings/facility-owners-and-managers/existing-buildings/save-energy`.

[35] *EnerNOC*.

[36] Orri Erling and Ivan Mikhailov. "RDF Support in the Virtuoso DBMS". In: *Networked Knowledge-Networked Media*. Springer, 2009, pp. 7–24.

[37] Gabe Fierro et al. "Mortar: an open testbed for portable building analytics". In: *Proceedings of the 5th Conference on Systems for Built Environments*. ACM. 2018, pp. 172–181.

[38] Inc Franz. *AllegroGraph: Semantic Graph Database*. `https://allegrograph.com/allegrograph/`. 2017.

[39] Flavius Frasincar, Alexandru Telea, and Geert-Jan Houben. "Adapting graph visualization techniques for the visualization of RDF data". In: *Visualizing the semantic web* 2006 (2006), pp. 154–171.

[40] Peter Fritzson and Vadim Engelson. "Modelica–A unified object-oriented language for system modeling and simulation". In: *European Conference on Object-Oriented Programming*. Springer. 1998, pp. 67–90.

[41] Sadayuki Furuhashi. "MessagePack: It's like JSON. but fast and small, 2014". In: *URL http://msgpack. org* (2017).

[42] Jingkun Gao, Joern Ploennigs, and Mario Berges. "A data-driven meta-data inference framework for building automation systems". In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 23–32.

[43] Inc Google. *Badwolf*. `https://google.github.io/badwolf/`. 2017.

[44] Siddharth Goyal, Herbert A Ingley, and Prabir Barooah. "Occupancy-based zone-climate control for energy-efficient buildings: Complexity vs. performance". In: *Applied Energy* 106 (2013), pp. 209–221.

[45] Thomas R Gruber. "Toward principles for the design of ontologies used for knowledge sharing?" In: *International journal of human-computer studies* 43.5-6 (1995), pp. 907–928.

[46] Sara Hachem, Thiago Teixeira, and Valérie Issarny. "Ontologies for the internet of things". In: *Proceedings of the 8th Middleware Doctoral Symposium*. ACM. 2011, p. 3.

[47] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. "SPARQL 1.1 query language". In: *W3C recommendation* 21.10 (2013).

[48] Andreas Harth and Stefan Decker. "Optimized index structures for querying rdf from the web". In: *Web Congress, 2005. LA-WEB 2005. Third Latin American*. IEEE. 2005, 10–pp.

[49] Philipp Heim et al. "RelFinder: Revealing Relationships in RDF Knowledge Bases." In: *SAMT* 5887 (2009), pp. 182–187.

[50] Dezhi Hong, Hongning Wang, and Kamin Whitehouse. "Clustering-based active learning on sensor type classification in buildings". In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM. 2015, pp. 363–372.

[51] Marco Jahn et al. "EnergyPULSE: tracking sustainable behavior in office environments". In: *Int. Conf. on Energy-Efficient Computing and Networking*. ACM. 2011, pp. 87–96.

[52] Deokwoo Jung et al. "Energytrack: Sensor-driven energy use analysis system". In: *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*. ACM. 2013, pp. 1–8.

[53] Srinivas Katipamula and Michael R Brambley. "Methods for fault detection, diagnostics, and prognostics for building systemsa review, part I". In: *Hvac&R Research* 11.1 (2005), pp. 3–25.

[54] Eamonn Keogh and Shruti Kasetty. "On the need for time series data mining benchmarks". In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '02* (2002), p. 102. ISSN: 13845810. DOI: `10.1145/775047.775062`. URL: `http://portal.acm.org/citation.cfm?doid=775047.775062`.

[55] *KGS Buildings*.

[56] Laura Klein et al. "Coordinating occupant behavior for building energy and comfort management using multi-agent systems". In: *Automation in construction* 22 (2012), pp. 525–536.

[57] Mario J. Kofler, Christian Reinisch, and Wolfgang Kastner. “A semantic representation of energy-related information in future smart homes”. In: *Energy and Buildings* 47 (2012), pp. 169–179.

[58] Jason Koh et al. “Plaster: an integration, benchmark, and development framework for metadata normalization methods”. In: *Proceedings of the 5th Conference on Systems for Built Environments*. ACM. 2018, pp. 1–10.

[59] Jason Koh et al. “Scrabble: transferrable semi-automated semantic metadata normalization using intermediate representation”. In: *Proceedings of the 5th Conference on Systems for Built Environments*. ACM. 2018, pp. 11–20.

[60] Andrew Krioukov et al. “A living laboratory study in personalized automated lighting controls”. In: ACM. 2011, pp. 1–6.

[61] Andrew Krioukov et al. “Building application stack (BAS)”. In: *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. ACM. 2012, pp. 72–79.

[62] Henrik Lange, Aslak Johansen, and Mikkel Baun Kjærgaard. “Evaluation of the opportunities and limitations of using IFC models as source of building metadata”. In: *Proceedings of the 5th Conference on Systems for Built Environments*. ACM. 2018, pp. 21–24.

[63] Ora Lassila and Ralph R Swick. “Resource description framework (RDF) model and syntax specification”. In: (1999).

[64] Jian Liang and Ruxu Du. “Model-based fault detection and diagnosis of HVAC systems using support vector machine method”. In: *International Journal of refrigeration* 30.6 (2007), pp. 1104–1114.

[65] Kamesh Madduri and Kesheng Wu. “Massive-scale RDF processing using compressed bitmap indexes”. In: *International Conference on Scientific and Statistical Database Management*. Springer. 2011, pp. 470–479.

[66] Alan Marchiori and Qi Han. “Using circuit-level power measurements in household energy management systems”. In: *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. ACM. 2009, pp. 7–12.

[67] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux Journal* 2014.239 (2014), p. 2.

[68] George A Miller. “WordNet: a lexical database for English”. In: *Communications of the ACM* 38.11 (1995), pp. 39–41.

[69] Natalie Mims et al. “Evaluation of U.S. Building Energy Benchmarking and Transparency Programs: Attributes, Impacts, and Best Practices”. In: (2017). DOI: `10.2172/1393621`. URL: `https://emp.lbl.gov/sites/default/files/lbnl%7B%5C_%7Dbenchmarking%7B%5C_%7Dfinal%7B%5C_%7D050417.pdf`.

[70] Mohamed Morsey et al. "DBpedia SPARQL benchmark–performance assessment with real queries on real data". In: *The Semantic Web–ISWC 2011* (2011), pp. 454–469.

[71] Inc Neo Technology. *Neo4j*. `https://neo4j.com/`. 2017.

[72] Thomas Neumann and Gerhard Weikum. "RDF-3X: a RISC-style engine for RDF". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 647–659.

[73] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[74] Natalya F Noy et al. "Creating semantic web contents with protege-2000". In: *IEEE intelligent systems* 16.2 (2001), pp. 60–71.

[75] Frauke Oldewurtel et al. "Use of model predictive control and weather forecasts for energy efficient building climate control". In: *Energy and Buildings* 45 (2012), pp. 15–27.

[76] *OWL Namespace*. `http://www.w3.org/2002/07/owl#`. 2018.

[77] Harshal Patni, Cory Henson, and Amit Sheth. "Linked sensor data". In: *Collaborative Technologies and Systems (CTS), 2010 International Symposium on*. IEEE. 2010, pp. 362–370.

[78] Mary Ann Piette, Sila Kiliccote, and Girish Ghatikar. "Field experience with and potential for multi-time scale grid transactions from responsive commercial buildings". In: (2014).

[79] Joern Ploennigs et al. "BASont-A modular, adaptive building automation system ontology". In: *IECON - 38th An. Conf. of IEEE Industrial Electronics Society*. IEEE. 2012, pp. 4827–4833.

[80] Joern Ploennigs et al. "Semantic models for physical processes in CPS at the example of occupant thermal comfort". In: *Industrial Electronics (ISIE), 2016 IEEE 25th International Symposium on*. IEEE. 2016, pp. 1061–1066.

[81] M. M. Polycarpou and A. J. Helmicki. "Automated fault detection and accommodation: a learning systems approach". In: *IEEE Transactions on Systems, Man, and Cybernetics* 25.11 (Nov. 1995), pp. 1447–1458. ISSN: 0018-9472. DOI: `10.1109/21.467710`.

[82] Marco Pritoni et al. "Short paper: A method for discovering functional relationships between air handling units and variable-air-volume boxes from sensor data". In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 133–136.

[83] Project Haystack. ***equipRef** definition*. `https://web.archive.org/web/20181213015204/https://project-haystack.org/tag/equipRef`.

[84] Project Haystack. *Implementing Project Haystack: Applying Haystack Tagging for a sample building*. `https://project-haystack.org/file/28/Reference-Implementation--Applying-Haystack-Tagging-for-a-Sample-Building.pdf`. 2018.

[85] *Project Haystack*. `http://project-haystack.org/`. 2018.

[86] *RDF Schema Namespace*. `https://www.w3.org/2000/01/rdf-schema#`. 2018.

[87] University of Rochester Utilities and Energy Operations Group Energy Management. *Building Automation System Design and Construction Standards*. `https://www.facilities.rochester.edu/central_utilities/documents/MasterStandards_11.2013.pdf`. 2013.

[88] Marko A Rodriguez. "The gremlin graph traversal machine and language (invited talk)". In: *Proceedings of the 15th Symposium on Database Programming Languages*. ACM. 2015, pp. 1–10.

[89] Craig Sayers. "Node-centric rdf graph visualization". In: *Mobile and Media Systems Laboratory, HP Labs* (2004).

[90] Jeffrey Schein et al. "A rule-based fault detection method for air handling units". In: *Energy and Buildings* 38.12 (2006), pp. 1485–1492.

[91] Michael Schmidt et al. "SP2Bench: a SPARQL performance benchmark". In: *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE. 2009, pp. 222–233.

[92] *Simulink Simscape*. `https://www.mathworks.com/products/simscape/features.html#multidomain-schematics/`. 2018.

[93] Jan Širokỳ et al. "Experimental analysis of model predictive control for an energy efficient building heating system". In: *Applied energy* 88.9 (2011), pp. 3079–3087.

[94] *SkyFoundry*.

[95] OpenLink Software. *Virtuoso*. `https://virtuoso.openlinksw.com/download/`. 2017.

[96] *SPARQL Query Language*. `https://www.w3.org/TR/rdf-sparql-query/`. 2018.

[97] Thanos G Stavropoulos et al. "Bonsai: a smart building ontology for ambient intelligence". In: *Proceedings of the 2nd international conference on web intelligence, mining and semantics*. ACM. 2012, p. 30.

[98] Markus Stocker et al. "SPARQL basic graph pattern optimization using selectivity estimation". In: *Proceedings of the 17th international conference on World Wide Web*. ACM. 2008, pp. 595–604.

[99] David Sturzenegger et al. "Semi-automated modular modeling of buildings for model predictive control". In: ACM. 2012, pp. 99–106.

[100] SYSTAP, LLC. *Bigdata Database Architecture Whitepaper*. `https://www.blazegraph.com/whitepapers/bigdata_architecture_whitepaper.pdf`. 2017.

[101] SYSTAP, LLC. *blazegraph*. `https://www.blazegraph.com/`. 2017.

[102] The Apache Software Foundation. "A free and open source Java framework for building Semantic Web and Linked Data applications". In: *https://jena.apache.org/* (2017).

[103] The Apache Software Foundation. *High performance Triple Datastore*. https://jena.apache.org/documentation/tdb/. 2017.

[104] The RDFLib Team. *RDFLib*. https://rdflib.readthedocs.io/en/stable/. 2017.

[105] TopQuadrant. *TopBraid Live*. http://www.topquadrant.com/products/topbraid-live/. 2017.

[106] *Turtle*. https://www.w3.org/TR/turtle/. 2018.

[107] U.S. Energy Information Administration. *Monthly Energy Review*. https://www.eia.gov/totalenergy/data/monthly/#consumption. 2018.

[108] U.S. Energy Information Administration. "User's Guide to the 2012 CBECS Public Use Microdata File". In: *Commercial Buildings Energy Consumption Survey (CBECS)* (May 2016), p. 33.

[109] Thomas Weng et al. "Managing plug-loads for demand response within buildings". In: ACM. 2011, pp. 13–18.

[110] Samuel R West et al. "Automated fault detection and diagnosis of HVAC subsystems using statistical machine learning". In: *12th International Conference of the International Building Performance Simulation Association*. 2011.

[111] Michael Wetter. "Co-simulation of building energy and control systems with the Building Controls Virtual Test Bed". In: *Journal of Building Performance Simulation* 4.3 (2011), pp. 185–203.

[112] Michael Wetter. "Modelica-based modelling and simulation to support research and development in building energy and control systems". In: *Journal of Building Performance Simulation* 2.2 (2009), pp. 143–161.

[113] Kesheng Wu et al. "FastBit: interactively searching massive data". In: *Journal of Physics: Conference Series*. Vol. 180. 1. IOP Publishing. 2009, p. 012053.

[114] Yuebin Yu, Denchai Woradechjumroen, and Daihong Yu. "A review of fault detection and diagnosis methodologies on air-handling units". In: *Energy and Buildings* 82 (2014), pp. 550–562.

[115] Chuan Zhang et al. "Knowledge management of eco-industrial park for efficient energy utilization through ontology-based approach". In: *Applied Energy* (2017).