Query Relaxation for Portable Brick-Based Applications

Imane Lahmam Bennani
ETH Zurich
limane@ethz.ch

Anand Krishnan Prakash
Lawrence Berkeley National Laboratory
akprakash@lbl.gov

Marina Zafiris
Lawrence Berkeley National Laboratory
mzafiris@lbl.gov

Lazlo Paul
Lawrence Berkeley National Laboratory
lpaul@lbl.gov

Carlos Duarte Roa
University of California Berkeley
cduarte@berkeley.edu

Paul Raftery
University of California Berkeley
p.raftery@berkeley.edu

Marco Pritoni
Lawrence Berkeley National Laboratory
mpritoni@lbl.gov

Gabe Fierro
Colorado School of Mines
National Renewable Energy Laboratory
gtfierro@mines.edu

ABSTRACT

Semantic metadata standards pave the way for interoperability by providing building operators and application developers with common schemes to describe building resources. Applications can query building metadata models to retrieve the set of entities and relationships they need to operate, instead of hard-coding references to specific points and objects from the underlying data sources.

Currently, querying such models requires the developer to be very specific when formulating queries in order to obtain meaningful answers (or any answer). The developer is inevitably expected to be familiar with the systems and components of the buildings being queried, as well as the schema used to represent them. The variety of buildings — both in the composition of their subsystems and in how they happen to be modeled — means that the developer will need to use multiple queries in order to retrieve necessary results. This is complex, time-consuming and error-prone.

To address this limitation, we investigate query relaxation as a technique to facilitate discovery of meaningful building resources in a collection of ontology-based buildings data. We evaluate our query relaxation approach over a set of Brick models and demonstrate its use in the context of real-world building applications.

CCS CONCEPTS

• **Information systems** → **Ontologies**; *Data encoding and canonicalization*; *Information extraction.*

KEYWORDS

RDF, Brick, SPARQL, Query Relaxation, Smart applications

BuildSys '21, November 17–18, 2021, Coimbra, Portugal

ACM ISBN 978-1-4503-9114-6/21/11.
https://doi.org/10.1145/3486611.3486671

ACM Reference Format:
Imane Lahmam Bennani, Anand Krishnan Prakash, Marina Zafiris, Lazlo Paul, Carlos Duarte Roa, Paul Raftery, Marco Pritoni, and Gabe Fierro. 2021. Query Relaxation for Portable Brick-Based Applications. In *The 8th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys '21), November 17–18, 2021, Coimbra, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3486611.3486671

1 INTRODUCTION

There has been a growing body of literature devoted to the development of advanced building analytics applications in recent years [10]. These applications have the potential to significantly reduce energy use, enhance occupant satisfaction, comfort, safety, and facilitate a proactive approach to building operations and maintenance [26]. Until a few years ago, these applications were developed on a one-off basis with limited portability across buildings [12]. The lack of *semantic interoperability* between digital representations of buildings and their data inhibits the goal of *portable applications* [29]. To address this issue, recent efforts have leveraged Semantic Web Technologies such as Resource Description Framework (RDF) [30], Web Ontology Language (OWL) [1] and SPARQL [2] to define standardized, machine-readable representations of building metadata. Examples of such endeavors include emerging semantic metadata schemas and standards such as Brick [7], Project Haystack [5], and ASHRAE Standard 223P [20]. These metadata schemas strive to provide uniform descriptions of building assets and subsystems and consequently reduce the effort required to author data-driven applications for buildings. However, this early body of work has been mostly concerned with capturing the complexity and heterogeneity of buildings and has not yet developed comprehensive tools that address the usability of these abstractions from the developer's perspective.

Without usable abstractions, the portability of applications will suffer. Currently, making effective use of these metadata schemas for portable applications requires the developer to have 1) a deep understanding of buildings and their subsystems 2) precise knowledge of the different concepts, relations and taxonomies defined by

the schema to describe those assets, 3) experience with a supported query language. This knowledge enables a developer to express queries against the metadata schema which return the information necessary to bootstrap or configure an application. However, it is difficult to make queries portable across many buildings. Differences in the structure and composition of building subsystems and differences in how modelers approach describing a building result in queries being successful for some buildings but failing for others. Hence, to foster the development of portable data-driven building applications, we need to transparently increase the portability of metadata queries so that they return relevant results on many different buildings.

1.1 Motivating Example

Metadata schemas like Brick define best practices and provide validation techniques to standardize representations of buildings using the schema. However, valid models for identical buildings may still vary due to design decisions made by modelers, motivated by differing target use cases, time constraints, and the modeler's experience and general interpretation of the structure of the building. This variability interferes with application portability.

Consider a "Rogue Zone Detection" (RZD) Application which identifies abnormal deviations between zone air temperature sensor measurements and their corresponding setpoints (control target) [27]. This deviation may result in excess energy consumption and compromise occupant thermal comfort. The RZD Application must first find the air temperature sensors and corresponding setpoints associated with a specific zone. These sensors and setpoints may be directly associated with the zone itself, a particular room, the terminal unit equipment which supplies heating/cooling to the zone, or indirectly associated through intermediate equipment such as a thermostat. These points may even have different type annotations within the digital model. This complicates the task of the application developer because they must be aware of the different possible arrangements to author a portable query.

The proposed method of query relaxation eliminates this complexity by automatically generalizing that query to execute on many different models. Figure 1 illustrates two alternative Brick models of the same HVAC system in a building: an air handling unit (AHU) supplying air to a variable air volume box (VAV) which serves an HVAC zone consisting of a room. While Model A was represented with a "zone" temperature sensor and setpoint associated with the VAV, Model B shows a generic air temperature sensor and setpoint associated to the zone directly. We will refer consistently back to this example and provide more details later in the paper. In other scenarios, the building systems themselves may be physically different, but a well formed query may still return usable results for some applications. For example, a naturally ventilated building with baseboard radiators may not have an air handling unit but will still typically have zone temperature sensors. Some applications could use the data from such a building as well as the example buildings in Figure 1.

1.2 Proposed Approach

To overcome modeling obstacles caused by limited building understanding and/or general variability in building models, we propose a method to generate "relaxed" versions of Brick queries from an original set of queries formulated by the developer. The key insight of our approach is that the developers only needs basic knowledge of building components and the Brick ontology to express their intentions and conditions. Query relaxation rewrites the original query so that it returns results on more buildings while preserving the intent of the query. Previous work has demonstrated such relaxation techniques being developed for both entity-relationship databases [14] and RDF datasets [25]. Query relaxation can enable developers to operate queries over buildings even if they lack knowledge of building structures and ontologies. Relaxing queries can also allow developers to write and leverage generic applications without having to consider all possible entries and configurations.

Our proposed approach to query relaxation operates using the formal definitions provided by the Brick ontology. We define a set of logical derivation rules that adhere to Brick taxonomy and semantic constraints. Relaxed queries can be generated by applying the set of derivation rules to an original SPARQL query in a simple manner.

1.3 Contributions

The contributions of this paper are three-fold. We provide:

(1) an algorithm for *relaxing* developer-provided queries so that they execute on a provided Brick model
(2) a similarity-based ranking model for assessing the relevance of relaxed Brick queries and for ranking them according to developers' preferences
(3) a functional proof-of-concept implementation for query relaxation, evaluated over a representative set of queries and real-world buildings and their Brick models that have been created by different modelers

The paper proceeds as follows. First, we provide a high-level overview of the Brick ontology, SPARQL query language, and prior work on query relaxation (§2). Then, we precisely define the query relaxation problem, and define a family of ontology-informed relaxation rules. We then discuss how to rank the relaxed queries created by our approach so the process returns the most semantically relevant queries with non-null results (§3). We describe the query relaxation algorithm and its implementation in a proof-of-concept system (§4), and we evaluate its performance and efficacy on a representative set of real-world queries and Brick models (§5).

2 BACKGROUND

2.1 The Brick Ontology

A *Brick model* is a directed, labeled graph describing the entities — physical, virtual, and logical "things" — in a building and the relationships between them. The structure of this graph is determined by the Brick ontology [7], a formal specification of the semantic representations of data sources in buildings and their context. Brick organizes entities using a hierarchy of *classes*. Classes are named sets of entities sharing similar properties which are formalized by the Brick ontology. Relationships between entities in a Brick model correspond to edges in the graph. Like classes, relationships have a label and a formal definition. They may also carry restrictions on what kinds of entities can have a particular relationship, and what kinds of entities can be the object of that relationship.

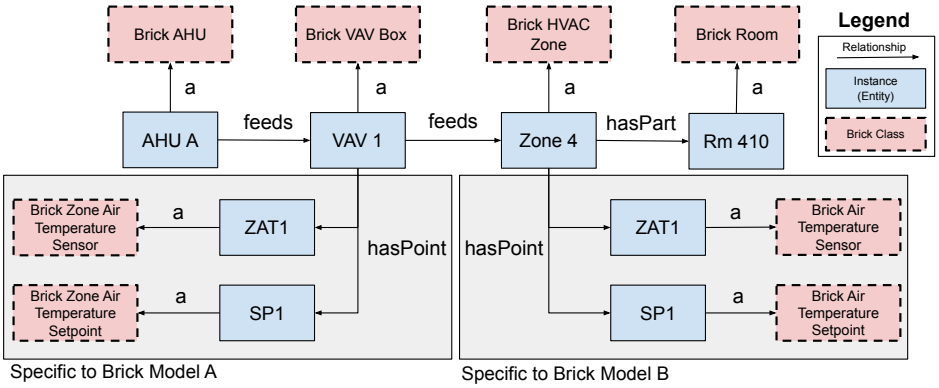

Figure 1: A simple Brick model illustrating a section of an air distribution system with a single zone and its temperature sensor and setpoint. These are associated either with a terminal unit (Model A) or the zone (Model B).

Brick models adhere to the RDF data model. RDF describes a directed, labeled graph as a series of 3-tuples called *triples*. A triple, written as $subject, predicate, object$ represents a directed edge between two nodes: the *subject* node has an edge *predicate* to another node *object*. The Brick ontology formally defines the meaning (*semantics*) and proper use of the classes and relationships using the OWL ontology language [1] and SHACL constraint language [3]. These languages provide an executable description of what statements are valid in a Brick model as well as what information can be automatically inferred about a Brick model. In normal usage, an external piece of software called a "reasoner" interprets the Brick ontology using the OWL and SHACL formal languages in order to validate the entities in a Brick model, and materialize any inferred properties or relationships. We will exploit these formal definitions to guide the process of query relaxation.

Figure 1 displays the two Brick models described in the motivating example. The types of the entities (AHU, VAV, etc.) are indicated by the `a` (`rdf:type`) relationship. The other relationships describe the connections between different entities in the HVAC system. The two models represent the same building, but differ in the way zone sensors and setpoints are described. Model A associates the two entities to the VAV Box using a `brick:hasPoint` relationship, while Model B associates them directly to the HVAC Zone. In addition, Model A has been modelled with more specific description of the points (i.e., `brick:Zone_Air_Temperature_Sensor`) while model B uses more generic classes (i.e., `brick:Air_Temperature_Sensor`), since the location of the sensor can be inferred from the relationship to the HVAC Zone.

2.2 SPARQL Queries

Software applications query Brick models in order to retrieve the metadata and configuration they need to operate. These queries are predominantly expressed in SPARQL [2], the W3C standard query language for the RDF data model. A SPARQL query (e.g., Figure 2) consists of a set of patterns describing a subgraph; evaluation of a query on a Brick model returns the subgraph matching the query predicate. These patterns are indicated by the `WHERE` clause of a SPARQL query. Each pattern follows the *subject*, *predicate*, *object* structure of an RDF triple; patterns may contain variables (indicated by a ? prefix) or relationships, entities, and other resources in the graph. A SPARQL `SELECT` clause defines the variables to be returned by the query.

SPARQL also defines a family of *property path operators* which augment how the RDF graph can be traversed during query execution. The `+` and `*` operators indicate that an arbitrary number of edges with the same label can be explored; for example the predicate `brick:feeds+` in a pattern would correspond to 1 or more `brick:feeds` edges between the subject and object. The `/` operator joins two predicates to indicate a sequence of edges through the graph; for example the predicate `brick:hasPart/brick:hasPoint` in a pattern would correspond to a `brick:hasPart` edge followed by a `brick:hasPoint` edge from the subject to the object. These expressive operators are instrumental in authoring SPARQL queries that return results on many different kinds of RDF graphs [15].

Consider the query in Figure 2, which retrieves the zone air temperature sensor and setpoint associated with entities that are downstream of an AHU. The `WHERE` clause first constrains each of the types of the variables to be returned: line 4 indicates that `?sen` should be bound to instances of the `brick:Zone_Air_Temperature_Sensor` class, for example. Line 7 indicates the topological relationship between the entity with whom the sensor and setpoint entities are associated, and the upstream AHU. Line 8 constrains the query to return pairs of temperature sensors and setpoints that are associated with the same entity. This SPARQL query is an example of what a developer might produce when writing a RZD Application. The query, as is, will only return the sensor and setpoint entities for Model A and will fail to produce any results on Model B (Figure 1). The query relaxation techniques discussed in this paper will automatically transform this query so that it returns the relevant entities on both models.

2.3 Query Relaxation and Related Work

Query relaxation is the process of rewriting a failing query so that it returns results on more inputs while preserving the intent of the original query. Such relaxation techniques have been developed for both entity-relationship databases [14] and RDF datasets [25].

```
PREFIX brick: <https://brickschema.org/schema/Brick#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
 ?sen   rdf:type   brick:Zone_Air_Temperature_Sensor .
 ?sp    rdf:type   brick:Zone_Air_Temperature_Setpoint .
 ?ahu   rdf:type   brick:AHU .
 ?ahu   brick:feeds ?thing .
 ?thing  brick:hasPoint ?sen .
 ?thing  brick:hasPoint ?sp .
}
```

Figure 2: A simple SPARQL query for retrieving zone air temperature sensors and setpoints connected to the same entity.

Broadly, these techniques exploit properties of the domain to reformulate the provided query in more generic terms. These properties may be provided by integrity constraints on the database or, in the case of RDF datasets, ontology statements that define the semantics of data in terms of first order logic. Without a formal ontology, query relaxation techniques must infer constraints from the instance data [28].

The work presented in this paper builds on existing work on query relaxation [11, 13, 21, 25] and applies these techniques to the building domain. *Cali et. al* [11] relax SPARQL queries using extensions to the SPARQL 1.1 query language to produce more results while querying an RDF dataset. The work presented in [13] relaxed queries based on user preferences in addition to existing constraints within the ontology itself and [25] used RDFS semantics to relax the queries. *Hogan et. al.* [21] relaxed RDF querying to return relevant answers in addition to perfectly matching answers using a distance measure between original and relaxed queries.

Metadata standards such as Haystack [5] and Brick [7] have enabled the development of portable applications and there has been significant research in the building domain to support similar work [8, 17]. [9, 16, 22] introduced tools to generate Brick models for buildings using both existing time-series data and metadata. The relaxation work presented in this paper allows for slight differences in the Brick representations generated and enables the queries to return a larger number of results. This is complementary to existing work on developing efficient query processors for Brick models, such as HodDB [15].

Existing query relaxation techniques have not been applied yet to the Brick ontology. Current tools developed to query Brick models also require exact knowledge of the query, the model, and the ontology itself. In this work, we investigate query relaxation in the context of data analytics applications for buildings [10]. Additionally, our work leverages the rich formal definitions provided by the ontology to relax queries from real data-driven building analytics applications. Brick is more complex than ontologies used in prior work [25] because it leverages both the OWL 2 RL ontology language and SHACL constraint language. In contrast, prior work has only concentrated on RDFS-based ontologies, which provide a subset of the expressive power of OWL 2 RL-based ontologies.

3 RELAXATION FRAMEWORK

Our proposed query relaxation procedure adopts a rule-based approach that is similar to prior work [18, 23, 24]. Rule-based query relaxation operates by repeatedly applying transformation rules to the query until it meets some termination conditions. We first provide mathematical preliminaries that underlie the query relaxation technique before formally defining the query relaxation problem. We then present the the different relaxation rules used in our framework (§3.3) and elaborate on the model used to rank the relaxed queries (§3.4) before we define the query relaxation algorithm (§4).

3.1 Preliminaries

Brick models are represented as RDF graphs. An RDF graph G is a set of triples $\{t_0, \ldots, t_m\}$, where each triple t_i is a 3-tuple

$$t_i \in (\mathcal{I}) \times (\mathcal{I}) \times (\mathcal{I} \times \mathcal{L})$$

where $\mathcal{I}$ is the set of IRIs [1] and $\mathcal{L}$ is the set of literals.

We adopt a simplified definition of a SPARQL query Q as a projection clause $s(Q) \in \mathcal{V}$ and a set $p(Q) \in P$ of *triple patterns*: $\{p_0, \ldots, p_n\}$. A triple pattern p_i is a 3-tuple

$$p_i \in (\mathcal{I} \times \mathcal{V}) \times (\mathcal{I} \times \mathcal{V}) \times (\mathcal{I} \times \mathcal{L} \times \mathcal{V})$$

where $\mathcal{V}$ is the set of query variables and $\mathcal{I}$ and $\mathcal{L}$ are defined as above. Executing a query Q on a graph G (annotated as $Q(G)$) produces a multiset of tuples with arity $|s(Q)|$, computed by taking the join of all intermediate relations produced by matching the triple patterns $p(Q)$ on the graph. The triple patterns may not match any subgraph of G, in which case the query returns no results.

3.2 Problem Definition

We address the generic problem of *conjunctive query relaxation* over RDF data in the context of smart building applications. We use the technique of *query relaxation* defined in §2.3 to enlarge the scope of a SPARQL query such that alternative results are returned over Brick models. A query Q is relaxed by applying a *relaxation rule* $r_i \in R$ to one or more of its triples, producing a new relaxed query Q'. A relaxation rule $r_i \in R$ is a function that takes a triple pattern p as an argument and returns one or several *relaxed triples* p'. $R\{r_1, r_2, \ldots, r_n\}$ is the set of relaxation rules. The patterns in the relaxed query Q' are given by

$$p(Q') = p(Q) - p_j + r_i(p_j)$$

where $p_j \in p(Q)$ is a pattern from the original query Q that has been rewritten using rule $r_i \in R$. If the application of the rule fails, i.e if the triple p cannot be relaxed anymore, the last relaxation of the triple pattern is returned. All patterns in a query are candidates for relaxation.

A *relaxation graph* G captures the transformation relationships between a query Q and the relaxed versions of the query. The nodes of the relaxation graph are relaxed queries $\{Q'_1, Q'_2, \ldots, Q'_n\}$ and the root node is the original query Q (Figure 3). Edges in the relaxation graph connect each query to the direct relaxations of the query. We define the *level* of a query in the relaxation graph as the total number of rules that have been applied to it with respect to the original query. In Subsection 4.1, we explain how to generate the relaxation graph of a query.

3.3 Relaxation Rules

Query relaxation can be expressed using *inference rules* derived from an ontology language such as RDFS [25]. Inference rules are logical functions which add implied information to an RDF graph based

[1] internationalized resource identifier: a generalized URL that indicates a named entity in the graph

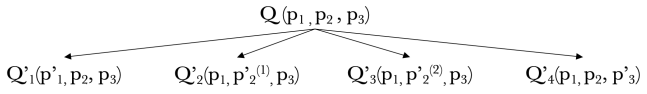

Figure 3: The relaxation graph of an initial query Q (level 1).

on the triples that exist in that graph and the ontology language which defines the semantics of those triples. A triple pattern p_i can be relaxed via the application of inference rules.

Consider the RDFS [19] ontology language which is used for query relaxation in [25]. Among other inference rules, RDFS defines the semantics of classes and instances such that a class is a named set, and an instance of the class is a member of that set. Another inference rule defines the semantics of subclasses: any instance of a class A is also an instance of all parent classes of A. In the context of query relaxation, any pattern p_i that matches instances of a class A could potentially be replaced by a modified pattern p_i' that matches instances of a *parent* class of A. This will necessarily match *at least* the same entities as the original pattern but may also match many more entities.

In this study, we build on prior work to formulate ontology-based relaxation rules that are adapted to the Brick ontology. We also consider additional rules not directly informed by the ontology such as the use of SPARQL property path operators to capture a wider variety of RDF graph structures. Our query relaxation process encompasses two types of relaxation: (1) class relaxation and (2) predicate relaxation. We now formalize the two types of relaxation rules and provide concrete examples of triple relaxation for each of them.

3.3.1 Entity Type Relaxation. Type relaxation uses class hierarchy information from the ontology to relax *type* conditions of Brick entities that are instances of Brick classes.

Relaxation Rule 1 (Superclass relaxation (sc))**.** Given a query pattern in the form (a, `type`, b) with (b, `subclassOf`, c), returns the relaxed query pattern (a, `type`, c). For example, the triple (`?sensor`, rdf:type, `brick:Zone_Temperature_Sensor`) can be relaxed to (`?sensor`, rdf:type, `brick:Air_Temperature_Sensor`), as `brick:Air_Temperature_Sensor` is a superclass of `brick:Zone_Air_Temperature_Sensor`.

3.3.2 Relationship Relaxation. This type of relaxation replaces the relationship (predicate) in a triple by another relationship that connects the subject entity to the object entity. Rule 2 below parses information from the ontology to derive other relationships whose domain or range includes the subject or object (respectively) in the triple. When applying this rule to a query pattern, we use the SHACL shapes given in Table 1 to enforce high-level domain/range restrictions and to validate the set of relaxed triples. Relaxation with Rule 3 can be performed without parsing the Brick ontology.

Relaxation Rule 2 (Domain/range-based predicate relaxation)**.** Given a query pattern in the form (a, pred_0, b) where pred_0 is the relationship between entities a and b, the rule returns the relaxed triple (a, pred_1 , c), where pred_1 is another relationship that connects a and b. For example, based on the SHACL shapes in Table 1, the triple (?ahu, brick:feeds , `?thing`) relaxes to the following set of triples: (?ahu, brick:hasPoint, `?thing`), (?ahu, brick:hasPart, `?thing`), (?ahu, brick:isPartOf, `?thing`), (?ahu, brick:isFedBy, `?thing`). This is because ahu is an instance of the root class `Equipment` and `thing` has no associated type in the query.

Relaxation Rule 3 (Predicate to property path relaxation)**.** Given a query pattern in the form (a, predicate, b), the rule returns the relaxed triple (a, predicate+ , c), where + is a property path operator. For instance, the triple (?ahu, brick:feeds , `?thing`) can be relaxed to (?ahu, brick:feeds+, `?thing`).

3.4 Ranking Model

Ranking is the process of determining which relaxed queries should be returned by the relaxation process. Relaxing an initial query Q_0 returns a potentially large set of relaxed queries $\{Q_1', Q_2', \ldots, Q_n'\}$ at each level of the relaxation graph. As more levels of the relaxation graph are explored, queries become more relaxed and potentially run on more Brick models. Moreover, the more patterns that are in the original query, the higher the number of the relaxed queries. For this reason, it is important to filter the set of possible relaxed queries down to those which are most likely to assist the portability of an application. Taking inspiration from *Huang et al.* [24], we present a relaxation-based ranking model that assumes that the more similar a relaxed query is to the original one, the more relevant the answers will be to the developer. We consider a $top-k$ model of query relaxation in which the k most highly ranked queries are returned by the process.

We use a similarity function to quantify the degree of closeness of a relaxed query Q' to the input query Q. Ranking is achieved by ordering the relaxed queries and their results in ascending order of similarity score. Given an initial query $Q(p_1, p_2,..., p_n)$ and a relaxed version of the query $Q'(p_1', p_2',..., p_m')$, the similarity score between the two queries is computed as follows:

$$\text{Sim}(Q, Q') = \prod_{i=1}^{n} w_i \cdot \text{Sim}\left(p_i, p_i'\right) \tag{1}$$

where $Sim(p_i, p_i')$ is the similarity between a triple pattern p_i in query Q and its relaxed version p_i' in Q', and $w_i \in (0, 1]$ is the weight of a triple pattern p_i that reflects the importance of p_i in Q.

Application developers determine the weight w_i of a triple p_i by considering the attributes of the original query they are willing to relax, given the application at hand. In this work, we simplify this task by restricting w_i to the set $W = \{0.1, 0.5, 1\}$ of values that can be assigned to each query pattern. Developers can assign a weight of 1 to the patterns in the query they are not willing to relax. Consequently, relaxed queries where those patterns have stayed unaltered are rewarded with a higher similarity score. Developers can assign a lower weight of 0.5 to the patterns they want to relax, such that relaxed queries where those triples have not been altered are penalized with a lower similarity score. Following the same logic, developers can also prioritize relaxing specific triples over others by choosing to assign an even lower weight of 0.1. If a developer does not provide weights, a default value of 1 is assigned to all the patterns in the query. In section 4.3, we show with an example how those weights can be determined and used in practice, in the context of a real-world building application.

Relationship	Definition	Domain	Range	Inverse
feeds	Subject conveys some media to the object entity in the context of some sequential process	Equipment Equipment	Equipment Location	isFedBy
hasPoint	Subject has a monitoring, sensing or control point given by the object entity	Equipment Location	Point Point	isPointOf
hasPart	Subject is composed – logically or physically – in part by the object entity	Equipment Location	Equipment Location	isPartOf

Table 1: High-level relationships enforced by SHACL Shapes in predicate relaxation.

Given a standard triple pattern $\boldsymbol{p}(s, p, o)$ and its relaxation $\boldsymbol{p'}(s', p', o')$, $Sim(p_i, p_i')$ is computed as follows:

$$\text{Sim}\left(\boldsymbol{p}, \boldsymbol{p'}\right) = \frac{1}{3} \cdot \text{Sim}\left(s, s'\right) + \frac{1}{3} \cdot \text{Sim}\left(p, p)'\right) + \frac{1}{3} \cdot \text{Sim}\left(o, o'\right) \quad (2)$$

To compute the similarity score in this paper, we consider individually (1) relaxed triples whose type has been relaxed, and (2) relaxed triples whose predicate (relationship) has been substituted.

For the former, a triple in the form (*a*, type, *b*) that assigns a class *b* to a subject *a* relaxes to a triple (*a*, type, *c*), where *c* is a superclass of *b* following the Brick class hierarchy. As only the object *o* of the original triple is relaxed, $Sim(s,s')$ and $Sim(p,p')$ are equal to 1, and the similarity between the two objects is defined by the similarity between the class *b* and its superclass *c*. Like in [24], we use *information content* as a measure of similarity between two classes. Information theory quantifies the information content of a class *c* as $I(c)$= -*log* Pr(*c*), where Pr(*c*) is here the probability of having an instance of class *c* in a Brick model. Given a class *c*, Pr(*c*) is defined as follows:

$$\Pr\left(c\right) = \frac{|\text{Instances}\left(c\right)|}{|\text{ Instances }|} \quad (3)$$

where *Instances*(*c*) is the number of entities in a Brick model that are instances of class *c* and *Instances* the total number of entities in the Brick model. Let c_1 and c_2 be two classes such as c_1 is a subclass of c_2. As c_2 encompasses c_1, the similarity between the two classes is given as follows:

$$\text{Sim}\left(c_1, c_2\right) = \frac{\text{I}\left(c_1, c_2\right)}{I\left(c_1\right)} = \frac{I\left(c_2\right)}{I\left(c_1\right)} = \frac{-\log \Pr\left(c_2\right)}{-\log \Pr\left(c_1\right)} \quad (4)$$

where $I(c_1,c_2)$ is the information content shared by the two classes c_1 and c_2. We now consider triples in the form (*a*, *p*, *b*) that connect two entities *a* and *b* with a relationship p. Applying the second relaxation rule from §3.3, the transformation occurs by substitution of the predicate with all the possible relationships that connect *a* and *b*. In that case, $Sim(s, s')$ and $Sim(o,o')$ in (2) are equal to one. This simple relaxation not guided by the ontology hierarchy can be regarded as relaxing the predicate to a variable, and we define the similarity between the predicate *p* and the relaxed predicate p' as 0. If instead the original triple relaxed using the predicate to property path relaxation rule, we assign a value of 1 to $Sim(p,p')$.

Once we compute the similarity score of all the triple patterns $p_i \in Q$ and of all their relaxed versions $p_i' \in Q'$, we can get the similarity score between the two queries Q and Q' using (2).

4 QUERY RELAXATION ALGORITHM

The relaxed queries and the corresponding relaxation graph are generated by a query relaxation algorithm. We describe the steps of the algorithm and its implementation, and provide an example rooted in the motivating RZD application.

4.1 Algorithm

The query relaxation algorithm applies the rules defined in §3.2 to produce a relaxation graph. Algorithm 1 presents the steps involved in the query relaxation process. The original query forms the root node in the relaxation graph. Applying relaxation rules to the triple patterns in this query generates a list of relaxations for each one of the triples. The Cartesian product of those lists defines the set of relaxed queries. The relaxed queries are organized into the graph by how many relaxations they contain. The diameter of the relaxation graph is limited by the maximum level of relaxation provided as input to the relaxation process. As defined in §3.2, the level in a relaxation graph is the total number of rules (or edits) applied to the original query.

To avoid presenting the developer with all the possible queries and their corresponding results, the algorithm prunes the relaxation graph to remove queries that return empty results when executed on a particular building. After pruning, the ranking model chooses the top-*k* queries using the developer-provided weights (described in §3.4) and computed similarity scores assigned to each of the remaining queries. The algorithm returns the top-*k* relaxed queries along with their similarity scores.

4.2 Implementation

The query relaxation algorithm along with its associated relaxation rules, utilities functions, and evaluation, were developed in Python3. We load version 1.2 of the Brick ontology using the RDFlib [6] package to perform the necessary extraction of rules from the ontology. This extraction uses the constraints on Brick relationships (Table 1) to inform the relaxation of relationships. The SHACL definitions provide more expressive domain and range constraints than what is provided by the RDFS ontology language.

The developer provides the algorithm with the Brick representation of a building along with the initial query to be relaxed and the number, *k*, of relaxed queries requested. The developer can also provide a maximum level of relaxation and weights for each triple. The query relaxation graph generated by the algorithm is stored using NetworkX [4], which facilitates exploration of the relaxed queries. The implementation, execution, evaluation scripts, and all the necessary packages related to the project can be found at the publicly accessible repository [2] .

[2] https://github.com/anandkp92/relaxed-brick-queries/

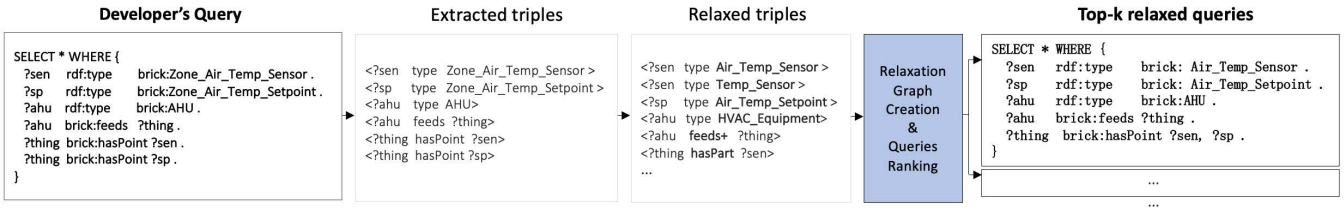

Figure 4: Query relaxation framework given a SPARQL query that retrieves air temperature sensors and setpoints connected to the same zone entity.

Algorithm 1: Query Relaxation Algorithm

Input: *query* (original query), $level_{limit}$ (max level of relaxation, optional), k (number of relaxed queries requested), w (set of user inputted weights for each triple in query), *building_model* (brick model of the building)
Result: set of at most k relaxed queries
extract triples from query;
for *triple in triples* **do**
  apply *relaxation_rules*(*triple*) ;
  save *relaxed_triples*
end
create empty graph $G_{relaxed}$;
add node *query* to $G_{relaxed}$ with *level*=0 ;
for *relaxed_query in permutations(relaxed_triples)* **do**
  $level_{relaxed\_query} = \sum_{i=1}^{number\_triples} level_i$;
  if $level_{relaxed\_query} < level_{limit}$ **then**
    if *node relaxed_query not in* $G_{relaxed}$ **then**
      add node *relaxed_query* to $G_{relaxed}$;
      set $level = level_{relaxed\_query}$;
    end
  end
end
for *node in* $G_{relaxed}$ **do**
  get *relaxed_query* from *node* ;
  res = execute_brick_query(building=*building_model*, query=*relaxed_query*) ;
  if *res is empty* **then**
    remove *node* from $G_{relaxed}$
  else
    assign *sim_score*(*original_query*, *relaxed_query*, w) to *node* ;
  end
end
return relaxed queries from top k nodes (based on *sim_score*) in $G_{relaxed}$

4.3 Running Example

To illustrate the query relaxation algorithm, consider the RZD application query in Figure 2 which retrieves zone air temperature sensors and setpoints connected to a same entity. The WHERE clause defines the query conditions using the following triple patterns: p_1(?sen, type, Zone_Air_Temperature_Sensor), p_2 (?sp, type, Zone_Air_Temperature_Setpoint), p_3(?ahu, type, AHU), p_4(?ahu, feeds, ?thing), p_5(?thing, hasPoint, ?sen), and p_6(?thing, hasPoint, ?sp).

Relaxed queries	Score
(?sen, type, Zone_Air_Temp_Sensor) (?sp, type, Zone_Air_Temp_Setpoint) (?ahu, type, AHU) (?ahu, feeds, ?thing) (?thing, hasPoint, ?sen) (?thing, **hasPoint+**, ?sp)	1.0
(?sen, type, Zone_Air_Temp_Sensor) (?sp, type, Zone_Air_Temp_Setpoint) (?ahu, type, AHU) (?ahu, feeds, ?thing) (?thing, **hasPoint+**, ?sen) (?thing, hasPoint, ?sp)	1.0
(?sen, type, Zone_Air_Temp_Sensor) (?sp, type, Zone_Air_Temp_Setpoint) (?ahu, type, AHU) (?ahu, **feeds+**, ?thing) (?thing, hasPoint, ?sen) (?thing, hasPoint, ?sp)	1.0

Figure 5: Top-3 relaxed queries and their similarity scores for case 1.

We select an arbitrary Brick model with a structure and resources similar to the Brick models presented in Figure 1 and we define a maximum relaxation level of 3. To compute the similarity scores after relaxation, we consider two case scenarios where different weights $w_i \in W = \{0.1, 0.5, 1\}$ are assigned to the triples p_i in the original query (as discussed in §3.4). Below we present the different weight assignments scenarios and discuss the results returned by the ranking model for the selected Brick model.

4.3.1 Case 1: Default value $w_i = 1$. In this case, we consider a scenario where a developer would like to limit relaxations on the query or does not provide weights to be assigned to the triples. A default weight value of 1 is attributed to all the triples in the query, and the similarity scores are computed accordingly. Figure 5 shows the top-3 relaxed queries returned for the selected building ranked in ascending order of similarity scores. In this case scenario, the top-3 relaxed queries returned all have a similarity score of 1. This is because only one predicate in one triple has been relaxed, and it was relaxed using the + property path operator. Because assigning a weight of 1 penalizes relaxations of triples (discussed in depth in §3.4), the top relaxed queries generated are the ones that return results on the Brick model while limiting edits on the original query. Referring to Figure 1, we can see that in this setting executing any of those relaxed queries on the Brick Model B would not return a result.

4.3.2 Case 2: $w(p_1), w(p_2) = 0.1$, $w(p_3), w(p_4) = 0.5$, and $w(p_5), w(p_6) = 1$. We now consider another setting where a developer is willing to relax a set of triples in the original query. We also consider that the developer has preferences towards relaxing

Relaxed queries	Score
(?sen, type, **Air_Temp_Sensor**) (?sp, type, **Air_Temp_Setpoint**) (?ahu, type, AHU) (?ahu, **feeds+**, ?thing) (?thing, hasPoint, ?sen) (?thing, hasPoint, ?sp)	0.45
(?sen, type, **Air_Temp_Sensor**) (?sp, type, **Air_Temp_Setpoint**) (?ahu, type, **HVAC_Equip**) (?ahu, feeds, ?thing) (?thing, hasPoint, ?sen) (?thing, hasPoint, ?sp)	0.35
(?sen, type, **Air_Temp_Sensor**) (?sp, type, **Air_Temp_Setpoint**) (?ahu, type, AHU) (?ahu, feeds, ?thing) (?thing, hasPoint, ?sen) (?thing, hasPoint, ?sp)	0.22

Figure 6: Top-3 relaxed queries and their similarity scores for case 2.

specific triples over others. As discussed in §1.1, the purpose of deploying a RZD application is to compare air temperature sensor measurements against their respective setpoints, and find extents of time where the measured temperature deviates significantly from its corresponding setpoint. This means that the query in Figure 2 can be extended to other types of temperature sensors. It can also be the case that different annotations have been used during modelling such that the air temperature sensor and setpoint are directly associated to a zone, as shown in Figure 1. In that sense, we decide to assign a lower weight of 0.1 to p_1 and p_2 in the original query, so that our ranking model prioritizes relaxation over these triples.

For the relaxed queries to keep the intent of the application, it is also crucial that the temperature sensors and setpoints are connected to the same entity. This means that relaxations over the triples p_5 and p_6 from the original query should be penalized, and we assign a weight of 1 to both triples. It is not as important for the application to keep the remaining triples p_1, p_2, p_3, and p_4 as is, and we attribute them a weight of 0.5. We can see in Figure 6 that the returned queries have lower similarity scores than in Case 1. This is because relaxation over some triples is now rewarded by lower weights assignments. As intended, p_5 and p_6 have not been altered while relaxation occurs for the other triples. In each of the relaxed queries, p_1 and p_2 have been altered to the more generic Brick class `Air_Temperature_Sensor`. Referring back to Figure 2, we can see that executing the relaxed queries returned on the Brick Model B would actually return results, as the scope of search has been extended.

With this heuristic approach, we see that it is possible for a developer to control the output of a relaxation process so that it fits the needs of an application through weights assignments. We also see that favoring some triple relaxations over others with the use of lower weights comes along with lower similarity scores. In this context, we can understand that the similarity score not only acts as an indication of the closeness of two queries, but also as a function that ranks queries according to developers preferences. In the next section, we evaluate the relaxation algorithm defined in §4 and discuss our results.

5 RESULTS AND DISCUSSION

In this section, we present an empirical evaluation of the query relaxation algorithm presented in §4. We evaluate our approach according to two metrics: (1) query relaxation overhead with respect to query execution time, and (2) portability of the query, measured in terms of additional buildings that return results to the relaxed queries.

	Min	Mean	Max
Number of triples	5	1011	8030
Number of classes	3	24	42
Number of relationships	2	526	4147
Number of entities	3	480	3870

Table 2: Brick models properties.

5.1 Experimental Setup

5.1.1 Dataset. We evaluated the query relaxation algorithm over 50 Brick-based representations of buildings extracted from the Mortar testbed [17]. All the models represent actual buildings created by different modellers, and cover a wide range of Brick classes, relationships, and literals queried by building-agnostic applications. Details on the distribution of the buildings' properties in the Brick models are given in Table 2.

5.1.2 Queries. Figure 7 shows the set of queries considered for the evaluation. All the queries presented are typical SPARQL queries drawn from popular applications designed for Brick models, obtained from the Mortar application suite and other research. For example, query Q_0 finds all sensors associated to the same entity that are instances of zone air temperatures sensors. We deliberately selected queries of different lengths (number of triples) and with simple types of clauses (e.g no `UNION` clause).

5.1.3 Preprocessing. As mentioned in §2.1, a reasoner has been applied to the Brick graphs before executing any of the relaxed queries. This preprocessing step allowed us to automatically generate additional implicit relationships in the Brick model (such as the superclasses and inverse relationships), thereby providing a much more complete and accurate building model for the relaxed queries to execute on.

5.1.4 Parameters. We defined for the experimental setup a maximum relaxation level of 3. This choice was guided by the observation of a reasonable increase of queries at that level. This aspect is discussed in depth in §5.2. To compute queries' similarity scores, we also defined a weight of 1 as a default value for all the triple patterns in the queries considered in this setup.

5.2 Query Relaxation Overhead

As a first step in our evaluation, we measured the time it takes to generate a relaxation graph (termed "relaxation time") for each of the queries selected, given a maximum relaxation level of 3. Table 3 summarizes our results. We observed reasonable relaxation times (<1 ms) for all the queries but Q7. As expected, the number of relaxed queries and relaxation time increase with the size of the query. For two queries of the same size (e.g Q_3 and Q_5), the number of relaxed queries and relaxation time can be different, as the content of the query also affects relaxation. The relaxation time measured in this evaluation does not include the time to execute

```
### Q0: Temp Sensors (Building Dashboard, Room Diagnostics)
SELECT * WHERE {
    ?sensor rdf:type brick:Zone_Air_Temperature_Sensor .
    ?sensor brick:isPointOf ?equip
}
### Q1: Temp Setpoints (Building Dashboard, Room Diagnostics)
SELECT * WHERE {
    ?sp rdf:type brick:Zone_Air_Temperature_Setpoint .
    ?sp brick:isPointOf ?equip .
}
### Q2: Airflow Sensors/Setpoints (Diagnostics)
SELECT ?sensor ?sp ?equip WHERE {
    ?sensor  rdf:type     brick:Air_Flow_Sensor .
    ?sp      rdf:type     brick:Air_Flow_Setpoint .
    ?sensor  brick:isPointOf ?equip .
    ?sp      brick:isPointOf ?equip .
}
### Q3: Airflow Sensors (Zone Ventilation Monitoring)
SELECT * WHERE {
    ?equip    rdf:type  brick:VAV .
    ?equip    brick:hasPoint  ?air_flow .
    ?air_flow rdf:type   brick:Supply_Air_Flow_Sensor .
    }
### Q4: VAV Enum (Building Dashboard)
SELECT ?vav WHERE {
    ?vav rdf:type brick:VAV .
}
### Q5: Spatial Mapping (Building Dashboard)
SELECT * WHERE {
    ?floor rdf:type brick:Floor .
    ?room  rdf:type brick:Room .
    ?room  brick:isPartOf+ ?floor .
}
### Q6: AHU Operation (Economizer Operation)
SELECT * WHERE {
    ?oat_damper a brick:Outside_Damper .
    ?pos        a brick:Damper_Position_Command .
    ?oat_damper brick:hasPoint ?pos .
    ?oat        a brick:Outside_Air_Temperature_Sensor .
}
### Q7: VAV Operation (Passing Valve Detection)
SELECT * WHERE {
    ?equip   rdf:type  brick:VAV .
    ?equip   brick:isFedBy  ?ahu .
    ?ahu     brick:hasPoint  ?upstream_ta .
    ?equip   brick:hasPoint  ?dnstream_ta .
    ?upstream_ta  rdf:type brick:Supply_Air_Temperature_Sensor .
    ?dnstream_ta  rdf:type brick:Supply_Air_Temperature_Sensor .
    ?equip   brick:hasPoint  ?vlv .
    ?vlv     rdf:type   brick:Valve_Command .
}
```

Figure 7: Selected Brick queries used in the evaluation.

Query ID	size of query	# relaxed queries	relaxation time(ms)
Q0	2	8	0.49
Q1	2	8	0.42
Q2	4	26	0.8
Q3	3	17	0.74
Q4	1	5	0.19
Q5	3	28	0.62
Q6	4	31	0.95
Q7	8	421	49.42

Table 3: Queries, number of relaxed queries, and relaxation overhead.

the queries on a Brick model for results retrieval. This aspect of the execution will be raised in the next section.

5.3 Query Portability Across Buildings

A second step in our evaluation consisted in assessing the efficacy of the proposed query relaxation algorithm in making queries more portable. To measure this, we tracked the number of additional buildings that return results on a relaxed query at different levels of relaxation (up to a maximum of 5). Figure 8 shows that after a relaxation level of 3, the number of buildings qualified increases for all the queries except Q5, which remained unaffected by any relaxation. This can be explained by the fact that Q5 refers to high-level classes (`brick:Floor` and `brick:Room`) and already contains a property path operator that would normally be added by a relationship relaxation rule.

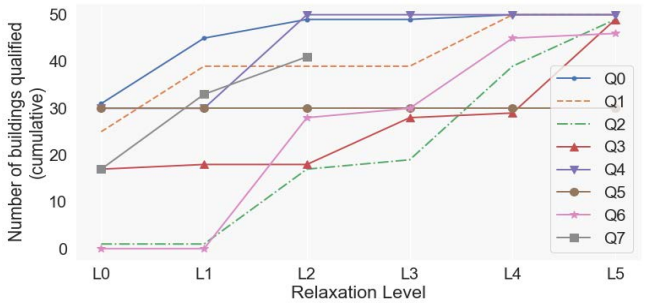

Figure 8: Number of unique buildings qualified at different levels of relaxation.

No data on buildings selectivity was obtained for Q7 beyond a relaxation level of 3. Because this query initially has 8 statements, even a three-level relaxation generates 421 queries (as shown in Table 3). In order to identify the relaxed queries that return results and to compute the number of buildings that qualify, we had to execute the relaxed queries across the 50 Brick models. This process took more than one hour, and as such we decided to stop the execution of the queries as it is not practical.

An increase in the size of an initial query yields more alternative queries and more opportunities to find a relaxed query that executes on more buildings. However, this is balanced by an increase in the time it takes to evaluate the relaxed queries and prune those which do not return results. This aspect of the evaluation highlights a limitation in our approach which is a heavy dependence on the performance of the query processor.

6 CONCLUSION AND FUTURE WORK

In this paper, we present a method for performing SPARQL query relaxation for buildings semantic models represented in Brick, and a similarity-based ranking model that can be adapted according to developers' preferences. This approach leverages the rich ontology defined in Brick, a graph-based metadata schema, to facilitate the discovery of resources in buildings by developers, and the writing of simple SPARQL queries which can be further extended with relaxation. We have evaluated our query relaxation algorithm over 8 queries drawn from popular applications, and executed the alternative queries generated on 50 Brick models from the Mortar testbed.

Results show that for all the selected queries, a relaxation graph can be computed within a reasonable amount of time. Although the relaxation time itself is negligible, executing the relaxed queries over a building to provide the developer with alternative queries that return data is more challenging. A more thorough evaluation

of the query execution time should be investigated and considered for optimization in future work. Future work may also consider query relaxation techniques which defer evaluation of the queries to reduce the coupling with the query processor, or perhaps eliminate this dependency altogether.

We have also showed that query relaxation efficiently increases query portability across buildings. This can ease the deployment of applications at a large scale. Future work should extend the query relaxation method presented to a broader set of modeling choices, variations in building configurations, and to more sophisticated queries. Finally, automatically inferring weight values from provided developers' preferences for the ranking model can also improve the usability of this approach.

Overall, this work improves the usability of query tools for semantic models, which is an important and understudied area of research. Developing better tools for the building industry practitioners and users of semantic models, such as Brick, is paramount to promote their adoption.

ACKNOWLEDGEMENTS

This research was funded by the US Department of Energy under Contract No. DE-AC02-05CH11231, Contract No. DE-AC36-08GO28308 and grant DE-EE0008681.

REFERENCES

[1] World Wide Web Consortium (W3C) 2012. *OWL 2 Web Ontology Language Document Overview (Second Edition) - W3C Recommendation 11 December 2012.* World Wide Web Consortium (W3C). http://www.w3.org/TR/owl2-overview/

[2] 2013. *SPARQL 1.1 Query Language.* https://www.w3.org/TR/sparql11-query/

[3] 2017. *Shapes constraint language (SHACL).* Technical Report. W3C. https://www.w3.org/TR/shacl/

[4] 2021. *NetworkX, Network Analysis in Python.* https://networkx.org/

[5] 2021. *Project Haystack.* https://project-haystack.org/

[6] 2021. *RDFLib.* https://github.com/RDFLib/rdflib

[7] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjærgaard, Mani Srivastava, and Kamin Whitehouse. 2016. Brick: Towards a Unified Metadata Schema For Buildings. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments* (Palo Alto, CA, USA) *(BuildSys '16)*. Association for Computing Machinery, New York, NY, USA, 41–50.

[8] Bharathan Balaji, Arka Bhattacharya, Gabe Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjærgaard, Mani Srivastava, and Kamin Whitehouse. 2016. Portable Queries Using the Brick Schema for Building Applications: Demo Abstract. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments* (Palo Alto, CA, USA) *(BuildSys '16)*. Association for Computing Machinery, New York, NY, USA, 219–220. https://doi.org/10.1145/2993422.2996411

[9] Arka A. Bhattacharya, Dezhi Hong, David Culler, Jorge Ortiz, Kamin Whitehouse, and Eugene Wu. 2015. Automated Metadata Construction to Support Portable Building Applications. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments* (Seoul, South Korea) *(BuildSys '15)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/2821650.2821667

[10] H. Burak Gunay, Weiming Shen, and Guy Newsham. 2019. Data analytics to improve building performance: A critical review. *Automation in Construction* 97 (2019), 96–109. https://doi.org/10.1016/j.autcon.2018.10.020

[11] Andrea Calì, Riccardo Frosini, Alexandra Poulovassilis, and Peter T. Wood. 2014. Flexible Querying for SPARQL. In *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*, Robert Meersman, Hervé Panetto, Tharam Dillon, Michele Missikoff, Lin Liu, Oscar Pastor, Alfredo Cuzzocrea, and Timos Sellis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 473–490.

[12] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David Culler. 2013. BOSS: Building Operating System Services. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 443–457. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/dawson-haggerty

[13] Peter Dolog, Heiner Stuckenschmidt, Holger Wache, and Jörg Diederich. 2009. Relaxing RDF queries based on user and domain preferences. *Journal of Intelligent Information Systems* 33, 3 (2009), 239.

[14] Shady Elbassuoni, Maya Ramanath, and Gerhard Weikum. 2011. Query relaxation for entity-relationship search. In *Extended Semantic Web Conference*. Springer, 62–76.

[15] Gabe Fierro and David E Culler. 2018. Design and analysis of a query processor for brick. *ACM Transactions on Sensor Networks (TOSN)* 14, 3-4 (2018), 1–25.

[16] Gabe Fierro, Jason Koh, Yuvraj Agarwal, Rajesh K. Gupta, and David E. Culler. 2019. Beyond a House of Sticks: Formalizing Metadata Tags with Brick. In *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation* (New York, NY, USA) *(BuildSys '19)*. Association for Computing Machinery, New York, NY, USA, 125–134. https://doi.org/10.1145/3360322.3360862

[17] Gabe Fierro, Marco Pritoni, Moustafa Abdelbaky, Daniel Lengyel, John Leyden, Anand Prakash, Pranav Gupta, Paul Raftery, Therese Peffer, Greg Thomson, and David E. Culler. 2019. Mortar: An Open Testbed for Portable Building Analytics. 16, 1, Article 7 (Dec. 2019), 31 pages. https://doi.org/10.1145/3366375

[18] Riccardo Frosini, A. Calì, A. Poulovassilis, and P. Wood. 2017. Flexible query processing for SPARQL. *Semantic Web* 8 (2017), 533–563.

[19] Ramanathan Guha and Dan Brickley. 2014. RDF Schema 1.1. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/

[20] Allen Haynes. 2018. ASHRAE's BACnet Committee, Project Haystack and Brick Schema Collaborating to Provide Unified Data Semantic Modeling Solution. https://www.ashrae.org/about/news/2018/ashrae-s-bacnet-committee-project-haystack-and-brick-schema-collaborating-to-provide-unified-data-semantic-modeling-solution

[21] Aidan Hogan, Marc Mellotte, Gavin Powell, and Dafni Stampouli. 2012. Towards Fuzzy Query-Relaxation for RDF. In *Proceedings of the 9th International Conference on The Semantic Web: Research and Applications* (Heraklion, Crete, Greece) *(ESWC'12)*. Springer-Verlag, Berlin, Heidelberg, 687–702. https://doi.org/10.1007/978-3-642-30284-8_53

[22] Dezhi Hong, Hongning Wang, Jorge Ortiz, and Kamin Whitehouse. 2015. The Building Adapter: Towards Quickly Applying Building Analytics at Scale. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments* (Seoul, South Korea) *(BuildSys '15)*. Association for Computing Machinery, New York, NY, USA, 123–132. https://doi.org/10.1145/2821650.2821657

[23] Hai Huang and Chengfei Liu. 2010. Query Relaxation for Star Queries on RDF. In *Web Information Systems Engineering – WISE 2010*, Lei Chen, Peter Triantafillou, and Torsten Suel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 376–389.

[24] Hai Huang, C. Liu, and Xiaofang Zhou. 2011. Approximating query answering on RDF databases. *World Wide Web* 15 (2011), 89–114.

[25] Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. 2008. Query Relaxation in RDF. In *Journal on Data Semantics X*, Stefano Spaccapietra (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 31–61.

[26] Hannah Kramer, Guanjing Lin, Claire Curtin, Eliot Crowe, and Jessica Granderson. 2020. Proving the Business Case for Building Analytics. https://buildings.lbl.gov/publications/proving-business-case-building

[27] Guanjing Lin, Marco Pritoni, Yimin Chen, and Jessica Granderson. 2020. Development and Implementation of Fault-Correction Algorithms in Fault Detection and Diagnostics Tools. *Energies* 13, 10 (2020). https://doi.org/10.3390/en13102598

[28] Ion Muslea. 2004. Machine learning for online query relaxation. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. 246–255.

[29] Marco Pritoni, Drew Paine, Gabriel Fierro, Cory Mosiman, Michael Poplawski, Avijit Saha, Joel Bender, and Jessica Granderson. 2021. Metadata Schemas and Ontologies for Building Energy Applications: A Critical Review and Use Case Analysis. *Energies* 14, 7 (2021), 2024. https://doi.org/10.3390/en14072024

[30] World Wide Web Consortium (W3C). 18 December 2020. *RDF Primer.* https://www.w3.org/TR/rdf-primer