

Design and Analysis of a Query Processor for Brick

GABE FIERRO, UC Berkeley

DAVID E. CULLER, UC Berkeley

Brick is a recently proposed metadata schema and ontology for describing building components and the relationships between them. It represents buildings as directed labeled graphs using the RDF data model. Using the SPARQL query language, building-agnostic applications query a Brick graph to discover the set of resources and relationships they require to operate. Latency-sensitive applications, such as user interfaces, demand response and model-predictive control, require fast queries — conventionally less than 100ms.

We benchmark a set of popular open-source and commercial SPARQL databases against three real Brick models using seven application queries and find that none of them meet this performance target. This lack of performance can be attributed to design decisions that optimize for queries over large graphs consisting of billions of triples, but give poor spatial locality and join performance on the small dense graphs typical of Brick. We present the design and evaluation of HodDB, a RDF/SPARQL database for Brick built over a node-based index structure. HodDB performs Brick queries 3-700x faster than leading SPARQL databases and consistently meets the 100ms threshold, enabling the portability of important latency-sensitive building applications.

This paper is an extension of a previously published work [16].

CCS Concepts: • **Information systems** → **Graph-based database models**; *Data structures*; *Information retrieval*;

Additional Key Words and Phrases: Smart Buildings, Building Management, Metadata, Graph Database, RDF, SPARQL

ACM Reference format:

Gabe Fierro and David E. Culler. 2018. Design and Analysis of a Query Processor for Brick. *ACM Trans. Sensor Netw.* 1, 1, Article 1 (January 2018), 25 pages.
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 INTRODUCTION

Modern buildings present a rich deployment opportunity for applications that take advantage of networked sensors and actuators to increase energy efficiency and comfort, as well as provide monitoring and fault diagnosis. While many such applications exist, the lack of a common description scheme, i.e. *metadata*, limits the portability of these applications across the heterogeneous building stock.

While several efforts address the heterogeneity of building metadata, these generally fail to capture the relationships and entities that are required by real-world applications [9]. This set of requirements drove the development of Brick [7], a recently proposed metadata standard for describing the set of entities and relationships within a building. Brick succeeds along three metrics:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.

1550-4859/2018/1-ART1 \$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

completeness (captures 98% of building management system data points across six real buildings), expressiveness (can capture all important relationships) and usability (represents this information in an easy-to-use manner).

Brick's goals of expressiveness and usability informed the choice of the RDF data model [24] and SPARQL query language [35] for representing and querying graphs, respectively. Initial work [7] showed that RDF/SPARQL fulfill Brick's requirements of description and representation, but did not address the question of how well suited these technologies are to fulfilling the "systems" requirements of Brick queries integrated into building applications. We focus on 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 [31]. 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) How can we leverage the characteristics of the Brick workload to design a query processor that does meet these requirements?

We begin with a brief overview of Brick, RDF and SPARQL, and then present 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.

2 BACKGROUND

This section provides a brief primer on the structure and usage of Brick and how it is realized using the RDF data model and SPARQL query language.

2.1 Brick Overview

Brick represents a building as a directed, labeled graph. Nodes (*entities*) represent equipment, sensors, spaces, timeseries streams or any other "thing" in a building. The names of nodes are drawn from Brick's class hierarchy. Edges represent the *relationships* between things and are named according to the minimal, multipurpose set of relationships defined by Brick.

Figure 1 shows the Brick graph for a simple example building; each node is labeled with its name or Brick class and each edge is labeled with a Brick relationship. The building consists of two adjacent rooms in an HVAC zone and conditioned by a variable air volume box (VAV) with a damper, which receives supply air from an air handling unit (AHU); one room contains a temperature sensor. The chain of feeds edges denotes that air passes from the AHU through the VAV and damper to the HVAC zone.

Brick helps mitigate heterogeneity, but also allows applications to understand salient structure. The particular sequence of equipment from an AHU to a zone differs from building to building. Because the "flow" has a consistent edge type (feeds), application developers can use the notion of "one or more feeds edges" to associate HVAC equipment with a zone without having to know the exact sequence. This is one way in which Brick allows queries to operate consistently despite differences in the structure of a building. This enables application portability while preserving the ability to recognize structure where important.

The example graph also captures part of the Brick class structure: each instance of a "thing" in a Brick graph has a type relationship to a node representing that class. Brick stores the class

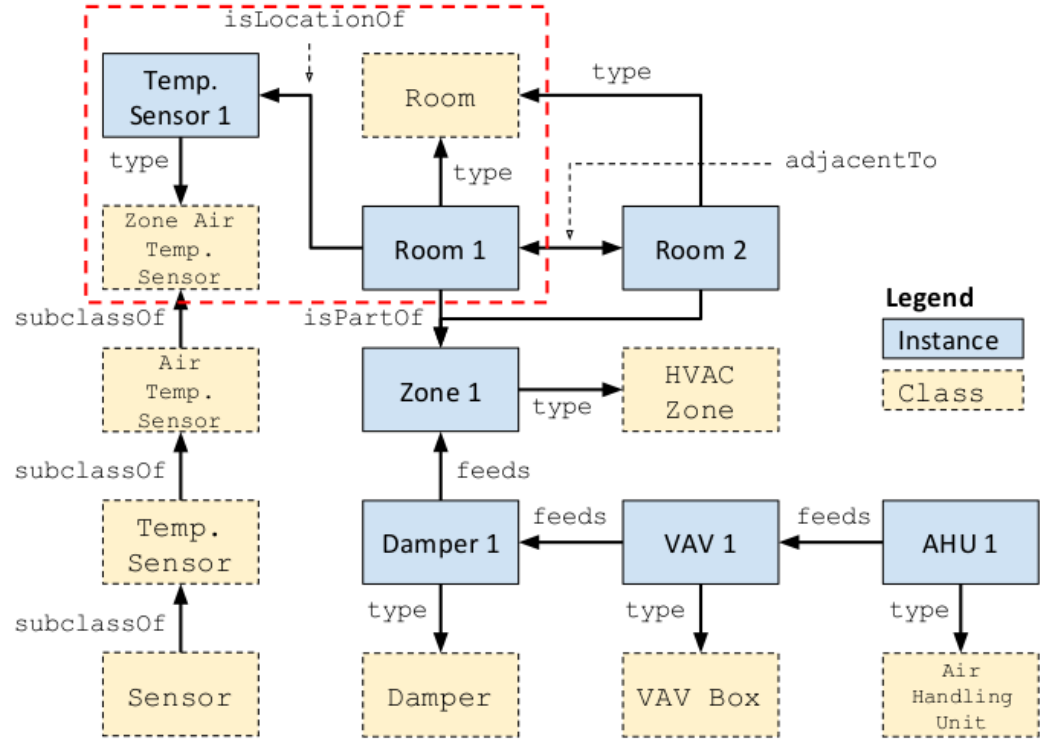


Fig. 1. Example Brick graph for a simple building, showing class instantiations and subclass declarations. Solid blue boxes make up the building, and dotted tan boxes represent the class hierarchy. Note the long chains of feeds and subClassOf relationships.

hierarchy itself in the graph using chains of subClassOf edges. The Brick class hierarchy helps account for uncertainty: the developer of a Brick model may not know the exact build or model of equipment in a building, and so can use a generic class (e.g. VAV) rather than a more specific (e.g. Trane VCCF Model VAV) class. Likewise, applications usually refer to generic parent classes, so Brick queries must have a way of specifying the semantics of the type system.

2.2 Brick Apps

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 sub-systems. 100ms is a common target for users to feel an interaction is “instantaneous” [31]. The dashboard application requires queries similar in structure to those generated by the interactive query explorer described in §6.

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. §6 describes in detail a data service that uses similar queries.

Room Diagnostics monitors the sets of sensors in each room to check for uncomfortable or unsafe conditions (such as high temperatures or CO₂ levels). The app queries the Brick model often

```

148 1  ### VAV Enum (Building Dashboard)
149 2  SELECT DISTINCT ?vav WHERE {
150 3    ?vav rdf:type brick:VAV .
151 4  }
152 5  ### Temp Sensors (Building Dashboard, Room Diagnostics)
153 6  SELECT DISTINCT ?sensor WHERE {
154 7    ?sensor rdf:type/rdfs:subClassOf* brick:Zone_Temperature_Sensor .
155 8  }
156 9  ### AHU Children (Building Dashboard)
157 10 SELECT DISTINCT ?x WHERE {
158 11   ?ahu rdf:type brick:AHU .
159 12   ?ahu bf:feeds+ ?x .
160 13 }
161 14 ### Spatial Mapping (Building Dashboard)
162 15 SELECT DISTINCT ?floor ?room ?zone WHERE {
163 16   ?floor rdf:type brick:Floor .
164 17   ?room rdf:type brick:Room .
165 18   ?zone rdf:type brick:HVAC_Zone .
166 19   ?room bf:isPartOf+ ?floor .
167 20   ?room bf:isPartOf+ ?zone .
168 21 }
169 22 ### Sensors In Rooms (Room Diagnostics)
170 23 SELECT DISTINCT ?sensor ?room
171 24 WHERE {
172 25   { ?sensor rdf:type/rdfs:subClassOf* brick:Zone_Temperature_Sensor . }
173 26   UNION
174 27   { ?sensor rdf:type/rdfs:subClassOf* brick:Discharge_Air_Temperature_Sensor . }
175 28   UNION
176 29   { ?sensor rdf:type/rdfs:subClassOf* brick:Occupancy_Sensor . }
177 30   UNION
178 31   { ?sensor rdf:type/rdfs:subClassOf* brick:CO2_Sensor . }
179 32   ?vav rdf:type brick:VAV .
180 33   ?zone rdf:type brick:HVAC_Zone .
181 34   ?room rdf:type brick:Room .
182 35   ?vav bf:feeds+ ?zone .
183 36   ?zone bf:hasPart ?room .
184 37   { ?sensor bf:isPointOf ?vav }
185 38   UNION
186 39   { ?sensor bf:isPointOf ?room }
187 40 }
188 41 ### VAV Relships (Building Dashboard)
189 42 SELECT DISTINCT ?vav ?x ?y ?z ?a ?b WHERE {
190 43   ?vav rdf:type brick:VAV .
191 44   ?vav bf:feeds+ ?x .
192 45   ?vav bf:isFedBy+ ?y .
193 46   ?vav bf:hasPoint+ ?z .
194 47   ?vav bf:hasPart+ ?a .
195 48 }
196 49 ### Grey Box (Automatic Grey Box Modeler)
197 50 SELECT DISTINCT ?vav ?room ?temp_uuid ?valve_uuid ?setpoint_uuid WHERE {
198 51   ?vav rdf:type brick:VAV .
199 52   ?vav bf:hasPoint ?tempsensor .
200 53   ?tempsensor rdf:type/rdfs:subClassOf* brick:Temperature_Sensor .
201 54   ?tempsensor bf:uuid ?temp_uuid .
202 55   ?vav bf:hasPoint ?valvesensor .
203 56   ?valvesensor rdf:type/rdfs:subClassOf* brick:Valve_Command .
204 57   ?valvesensor bf:uuid ?valve_uuid .
205 58   ?vav bf:hasPoint ?setpoint .
206 59   ?setpoint rdf:type/rdfs:subClassOf* brick:Zone_Temperature_Setpoint .
207 60   ?setpoint bf:uuid ?setpoint_uuid .
208 61   ?room rdf:type brick:Room .
209 62   ?tempsensor bf:isLocatedIn ?room .
210 63 }

```

Fig. 2. The set of SPARQL queries used in real-world Brick apps, used here for benchmarking RDF databases in §3. 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.

```

1 @prefix bf: <https://brickschema.org/schema/1.0.1/BrickFrame#> .
2 @prefix brick: <https://brickschema.org/schema/1.0.1/Brick#> .
3 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
4 @prefix bldg: <http://brickuniversity.edu/buildings/BuildingABC#> .
5 bldg:Room_1 rdf:type brick:Room .
6 bldg:Temp_Sensor_1 rdf:type brick:Zone_Air_Temperature_Sensor .
7 bldg:Room_1 bf:isLocationOf bldg:Temp_Sensor 1 .

```

Fig. 3. The set of triples for the highlighted part of the graph in Figure 1, using the Turtle format for RDF data [8].

Figure 2 shows the queries constituting these applications. Other categories of applications that can benefit from fast metadata queries are fast demand response [34], model-predictive control, and online fault detection and diagnosis. [7] and [9] present more comprehensive lists of metadata-driven applications.

2.3 RDF Data Model

Brick graphs are specified using the RDF data model [24]. This is a syntax-independent way of describing directed, labeled graphs as a set of *triples*. A triple is a 3-tuple <subject, predicate, object> that states that an entity *subject* has a relationship *predicate* (directed edge) to an entity *object*. A Brick model for a building consists of a set of triples.

All entities and relationships exist in a namespace, identified by a URI. For example, the Brick entity namespace is <https://brickschema.org/schema/1.0.1/Brick#>. Namespaces are usually abbreviated to a prefix e.g. brick:, so we could represent the AHU class in the Brick namespace as brick:AHU.

The Brick ontology makes prevalent use of the standard rdf [1] and rdfs [2] ontologies (see Table 1). The RDF data model can also represent literal values, which Brick uses to store information such as coordinates and pointers to timeseries streams (§ 6).

Abbreviation	Full Namespace	Usage
rdf:	<http://www.w3.org/1999/02/22-rdf-syntax-ns#> [1]	Used for instantiation (rdf:type)
rdfs:	<http://www.w3.org/2000/01/rdf-schema#> [2]	Used for subclasses (rdfs:subClassOf)
brick:	<https://brickschema.org/schema/1.0.1/Brick#>	Used for Brick classes (e.g. brick:Temperature_Sensor)
bf:	<https://brickschema.org/schema/1.0.1/BrickFrame#>	Used for Brick relationships (e.g. bf:feeds)

Table 1. The ontologies used in this paper along with common namespace abbreviations.

2.4 SPARQL Query Language

Applications query a Brick model to retrieve the particular set of entities, relationships and literals they need to operate. Queries use SPARQL (SPARQL Protocol and RDF Query Language) [35] to define a set of *patterns* that constrain the set of RDF terms returned from the graph.

SPARQL queries consist of SELECT and WHERE clauses. The WHERE clause consists of a set of patterns that use the RDF <subject, predicate, object> triple structure, but any of the terms may be a variable (indicated by a ? prefix). The results of a query are the set of RDF terms matching the variables in the SELECT clause.

Consider the VAV Enum query from Figure 2: the WHERE clause defines a single variable ?vav which the pattern constrains to be all entities that have an edge rdf:type to the node brick:VAV representing the Brick VAV class. This lists all instances of the VAV class in a building.

The SPARQL 1.1 standard [21] expands the base language to support more flexibility in these patterns. For Brick the most important of these are the *property path operators*, which include:

- / matches a sequence of paths (bf:feeds/bf:hasPart)
- * matches a chain of zero or more edges (rdfs:subClassOf*)
- + matches a chain of one or more edges (bf:feeds+)
- ? matches zero or one edges (rdfs:subClassOf?)
- | matches one of a set of paths (bf:hasPart|bf:hasPoint)

Brick queries make extensive use of these operators because they enable query authors to remain somewhat agnostic to exact sequences of edges, which makes queries more portable to different buildings. However, this additional expressive power comes at the cost of query evaluation time. §4 discusses this in depth.

2.5 Typical Brick Queries

Figure 2 shows the set of representative queries used for benchmarking in §3. 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 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). HodDB addresses this challenge using a path index (§5).

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).

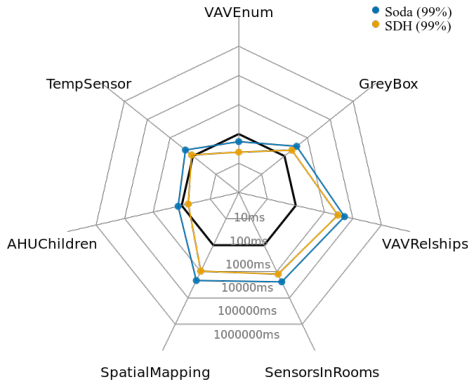
SensorsInRooms associates a family of sensors with a room, using the room's HVAC zone and VAV information. The query makes heavy use of UNION to select the appropriate sensor classes.

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.

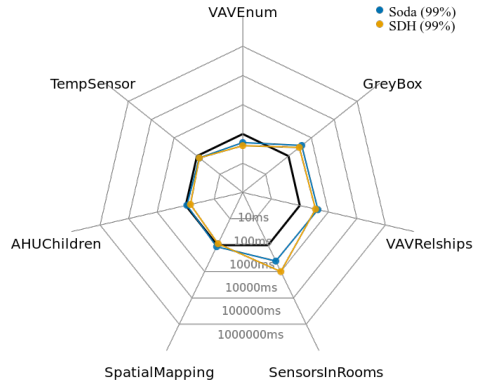
GreyBox identifies, for each room in a building, a minimal set of sensor streams (identified by a UUID) that can be used to train a simple grey box thermal model.

3 RDF DATABASE COMPARISON

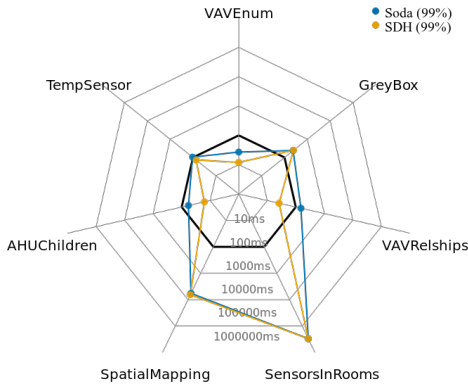
We evaluate the performance of several popular SPARQL databases on three Brick graphs using a set of seven queries used by real Brick applications requiring low and predictable latency; we target a 99th percentile query latency of <100ms. §4 characterizes the requirements of the Brick workload in more detail.



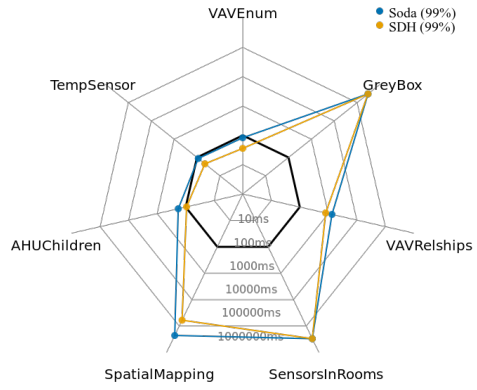
(a) Allegrograph result for Soda and SDH



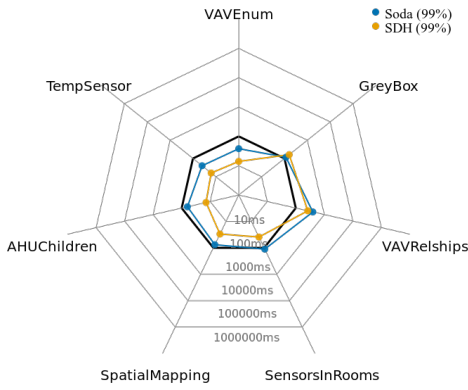
(b) Blazegraph result for Soda and SDH



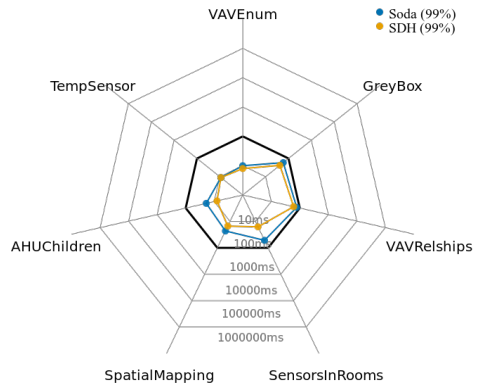
(c) Apache Jena result for Soda and SDH



(d) RDFLib result for Soda and SDH



(e) Virtuoso result for Soda and SDH



(f) HodDB result for Soda and SDH

Fig. 4. 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.

3.1 RDF Databases

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 [43] 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 [22] (explained in §4), which is a common implementation approach.

Blazegraph [41, 42] 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 supports geospatial data.

RDF-3X [30] 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.

RDFLib [45] 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 [4, 17] 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 [15, 39] 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 [46], for which we could not obtain a copy, and the RDF extension [25] to the FastBit [47] storage system, which has no available implementation. Further, our evaluation focuses on available RDF databases that implement the SPARQL query language. This disqualifies several other RDF and graph databases (such as Cayley [12], Dgraph [14], Badwolf [20] and Neo4j [29]), which implement alternative graph query languages such as Grem-lin [36] and Cypher. While it can be shown that these other languages can express many of the same relations as SPARQL, SPARQL is the W3C recommended language for querying RDF data and is the recommended query language by the Brick authors. An evaluation of other query languages is a subject for future work.

3.2 Experimental Setup

We evaluate the Brick workload 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 6 (discussed later).

Our evaluation consists of running the set of Brick queries from Figure 2 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.

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
AHUCChildren	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 2. 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
AHUCChildren	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 3. 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
AHUCChildren	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 4. Query latency distribution for a large building (SDH).

We develop a simple test harness¹ to dispatch each benchmark query against each database and measure the time in milliseconds from the time the query was dispatched to the time 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. 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 [26] for consistent and replicable execution of each database. As development of HodDB progresses, we will continue to improve and expand the test

¹https://github.com/gtfierro/brick_database_eval

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 [6]. 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 a server with a 3.5 GHz Intel Xeon E5-1650 CPU; all databases were backed by a dedicated SSD.

3.3 Brick Query Performance

Tables 2, 3 and 4 show the mean, standard deviation and 99th percentile latencies for each of the benchmark queries (Figure 2) over the three Brick buildings from Table 6. We report the distribution for completeness, but 99th percentile latency is the key metric. We defer discussion of the last column (HodDB) until §5. 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 4 visualizes the benchmark 99th 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 2), but substantially degraded performance on the two larger buildings (Tables 3 and 4). Only Allegrograph, Blazegraph and Virtuoso are able to complete each query on the two large Brick buildings in less than 5 minutes². 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, the two primary factors are the number of patterns in a query and use of the `*` and `+` property path operators (§4.2).

Increasing the number of patterns in a query corresponds to increased pressure on the “join” mechanism in the executing database, which tends to be one of the dominating factors in query performance [11, 28]. 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. To quantify this effect, we run the AHUChildren query applying different property path operators to the `bf:feeds` term. Table 5 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.

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. This “pattern amplification” happens because `*` and `+` can force a database to resort to slower graph traversal

²In 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 5. Effect of property path operators on query execution time. All times are in milliseconds. This microbenchmark was run against the Soda Brick model.

	RDF Dataset	Triples	Nodes	Edge Types	Avg Out Degree per Edge Type
Real	Infobox [10]	30,024,092	9,741,482	2063	.0015
	Wordnet [27]	8,574,806	2,487,208	64	.0539
	Sensor [33]	185,950	86,580	12	.179
Syn.	SP2B [38]	7,442	4,800	57	.0272
	BSBM [11]	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 6. Graph properties of some published RDF datasets and three representative Brick models.

rather than relying on optimized joins between its B-tree indices. The classic YARS-style index (§) 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.

Unsurprisingly, our performance analysis above suggests that “join” performance is a primary component of SPARQL query execution time. 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 additional edge during the evaluation of patterns involving /, + or * query operators. 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 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.

4 BRICK WORKLOAD

To understand the requirements of a query processor for Brick, we characterize the graphs and queries that constitute a typical Brick workload, and discuss how these properties affect the performance of state-of-the-art query processors.

4.1 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).

Table 6 compares the size and density of several real-world datasets (DBPedia Infobox [10], LinkedSensor [33] and Wordnet [27]), synthetic datasets (BSBM [11] and SP2B [38]) 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.

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

Table 7. Properties of the benchmark SPARQL queries

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.

4.2 SPARQL Features

Brick queries only require a subset of features defined by the SPARQL 1.1 specification [21]. These features are characterized by how they allow a query to express uncertainty in the structure of the graph. This is vital for the Brick workload because queries are typically written to a family of graphs rather than for a specific instance, so there is a degree of expected heterogeneity. In contrast, many RDF queries only target a specific graph.

The heading of Table 7 shows the SPARQL 1.1 features Brick requires. UNION and | allow queries to express the notion of “or”. The property path operators +, *, ?, and / allow flexible matching of arbitrary-length chains of relationships. Matching chains of relationships is necessary when the query author does not know how many edges separate two nodes, but knows the kinds of relationships involved.

For example, it is important in Brick to be able to write a query involving a generic superclass (such as “Temperature Sensor”) even though the actual nodes in the graph may be instances of a more specific subclass. To express subtype polymorphism in SPARQL, Brick queries often involve constructions such as

```
1 ?sensor rdf:type/rdfs:subClassOf* brick:Temperature_Sensor .
```

which matches 0 or more rdfs:subClassOf edges (expressing subtyping), followed by one rdf:type edge (expressing an instance).

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 we could rewrite the AHUChildren query to explicitly search for bf:feeds paths of length 1 and of length 2, but this would limit the portability of the query and require prior knowledge of the graph structure. The Brick 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 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 §5, 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 .

4.3 RDF Index Structures

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 [22], 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 (subject, predicate and object) 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 [22], RDFLib [45], RDF3X [30], Blazegraph [41] and the TDB engine behind Jena [44] 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 2: 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 [22]. This motivates the design of a query processor designed specifically for Brick graphs.

5 DESIGN OF HODDB

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

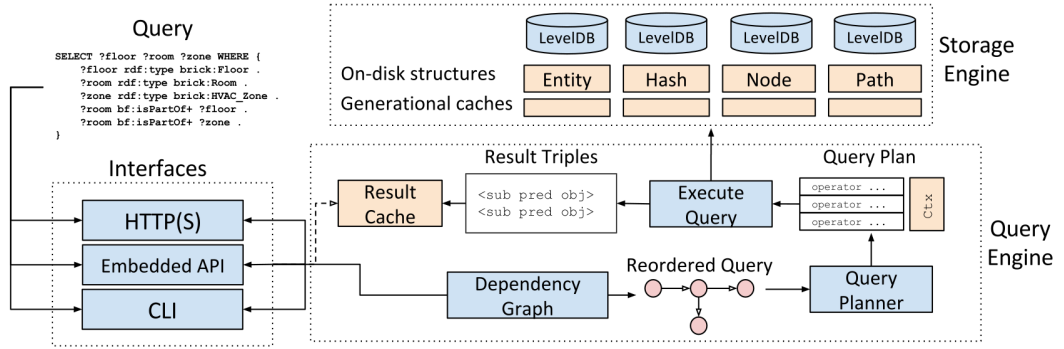


Fig. 5. Architecture of HodDB

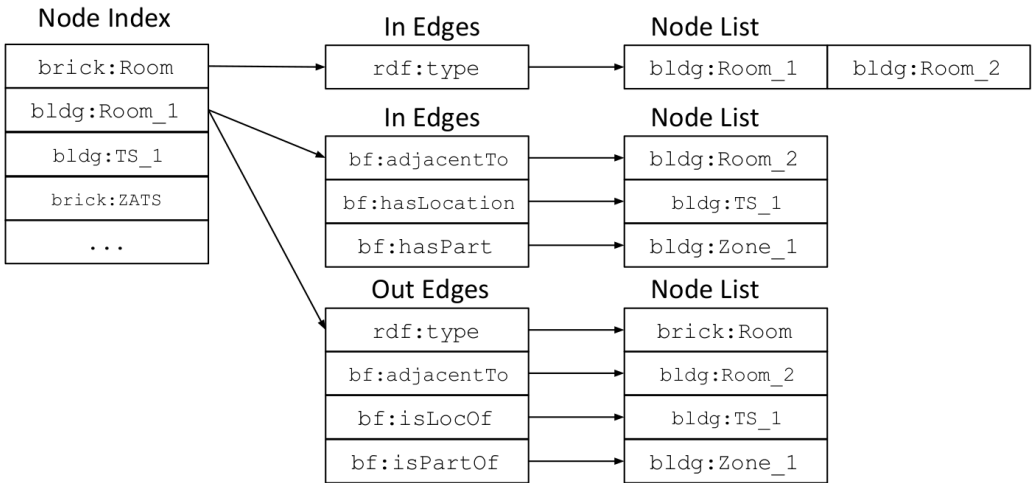


Fig. 6. Node index structure for the highlighted part of Figure 1.

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) restrict supported data types to strings.

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 5.

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.

5.1 Storage Engine

HodDB stores the RDF triples constituting a Brick graph in a family of index structures, each of which is backed by an instance of LevelDB³, a popular embedded key-value database with support for range queries and transactions. All HodDB indices are built over a key-value abstraction.

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 5) 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.

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 value contains 2 MsgPack [19]-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 6 shows this structure for the `brick:Room` and `bldg:Room_1` nodes in Figure 1. 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 elaborating preprocessing step we applied to other RDF databases.

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. In denser graphs that have a higher average fanout per node, like Brick models (Figure 6), 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 2, 3 and 4). The **predicate index** is similar in structure to the node index, but uses predicate/edge hashes as keys.

³We use a Go port: <https://github.com/syndtr/goleveldb>

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

5.2 Query Engine

We now describe HodDB’s query evaluation engine, depicted in Figure 5. 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 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 SPARQL patterns can severely impact execution time [40]. 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 SPARQL pattern and associates an operator with each pattern according to the positioning of variables in that pattern (Table 8). An operator is a small piece of code that takes a SPARQL 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

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. 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.

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. HodDB joins the output of each operator with the context relation to form the results of the query.

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 SPARQL 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 data 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.⁴ We disabled the result cache for all measurements of HodDB, but it generally returns results in <1ms on a cache hit.

Implementation: HodDB is free and open-source⁵ and implemented in Go [3]⁶, a modern compiled programming language with builtin concurrency primitives: goroutines (extremely lightweight "threads" of execution scheduled in userland) and channels (atomic FIFO queues with optional buffers). These primitives allow HodDB to support many concurrent queries and scale to several cores with minimal locking infrastructure. Benchmarking how many queries-per-second HodDB supports is a subject of future work.

One challenge in working with Go is dealing with garbage collection (GC). Care has been taken in the implementation of HodDB to use object pools to reduce allocation, but HodDB still experiences occasional GC pauses that can increase query latency by 200%. Current development on HodDB seeks to address this issue.

5.3 Evaluation

Microbenchmarks: Referring back to Table 5, HodDB's path index means that property path operators only induce a 38% overhead on query execution time. Table 9 compares disk usage for

⁴Which is how MySQL's optional result cache works

⁵<https://hoddb.org/>

⁶Go version 1.9.2 at time of writing

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 9. 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 10. 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.

each graph for each database. HodDB does not apply specialized compression techniques, but we can conclude that HodDB's index structure does not raise any disk utilization concerns.

HodDB's design decisions target small, dense graphs that typify Brick models. An obvious question is how well HodDB 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 2). 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 10 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 §3 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 [11] or [38] 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 2, 3 and 4; 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.

6 NEW APPLICATIONS

As an interactive-speed SPARQL query processor, 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.

6.1 Interactive Query Visualizer

Visualization is a common technique for making sense of RDF graphs [13, 18, 23, 37]. 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 [32] 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 v0.5.5⁷ and onward ship with a web frontend implementation of this method. Figure 7 is a screenshot of a sample interaction. The user begins with the `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 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

⁷<https://github.com/gtfierro/hod/releases/tag/v0.5.5>

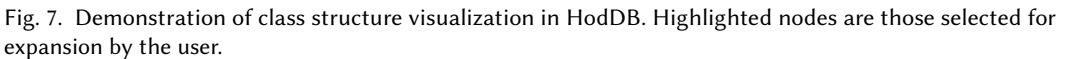


Fig. 8. Autogenerated query from the interaction in Figure 7. Variable names are autogenerated. The SELECT clause contains the variables representing nodes selected by the user. HodDB drops the autogenerated terms containing 3 variables when returning the query to the user

- 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.

ACM Transactions on Sensor Networks, Vol. 1, No. 1, Article 1. Publication date: January 2018.

6.2 Integrating Brick with HodDB

In addition to enabling new modes of visualization, HodDB's speed also makes it suitable for embedding in applications. These 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. The upshot is applications can leverage the contents and structure of a Brick model for their own operation. We will explore two services that use HodDB to tightly integrate with a Brick model.

```

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

```

Fig. 9. Example of augmenting a Brick model representation of a temperature sensor with a 36-byte UUID used for historical data access.

6.2.1 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.

MDAL makes use of existing monitoring infrastructure⁸ 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 9).

MDAL queries describe the composition of the desired dataset; Figure 10 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 10, 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.

6.2.2 Brick-driven Scheduler. A scheduler is a simple example of how HodDB can be a configuration point 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 11

⁸Provided by the XBOS Project: <https://docs.xbos.io/>

```

1030 1 {
1031 2   "Composition": ["internaltemp", "externaltemp"],
1032 3   "Selectors": ["mean", "mean"],
1033 4   "Variables": [
1034 5     {
1035 6       "Name": "internaltemp",
1036 7       "Definition": "SELECT ?uuid WHERE { ?sensor rdf:type/rdfs:subClassOf* brick:Temperature_Sensor .
1037 8         ?sensor bf:uuid ?uuid . ?sensor bf:isLocatedIn ?r . ?r rdf:type brick:Room }";
1038 9       "Units": "C",
1039 10    },
1040 11    {
1041 12       "Name": "externaltemp",
1042 13       "Definition": "SELECT ?uuid WHERE { ?sensor rdf:type/rdfs:subClassOf* brick:Outside_Temperature_Sensor .
1043 14         ?sensor bf:uuid ?uuid }";
1044 15       "Units": "C",
1045 16    },
1046 17  ],
1047 18  "Time": {
1048 19    "T0": "2017-08-01 00:00:00",
1049 20    "T1": "2017-08-31 00:00:00",
1050 21    "WindowSize": "1h",
1051 22    "Aligned": "True",
1052 23  },
1053 24  }

```

Fig. 10. Sample MDAL query retrieving internal and external temperature data for a deployment resampled to 1-hour means.

```

1051 1 while True:
1052 2   thermostats = hodb.run_query("""
1053 3   SELECT ?tstat ?api ?zone WHERE {
1054 4     ?tstat rdf:type brick:Thermostat . ?tstat bf:uri ?api .
1055 5     ?tstat bf:controls/bf:feeds+ ?zone .
1056 6     ?zone rdf:type brick:HVAC_Zone
1057 7   }
1058 8   """)
1059 9   for thermostat in thermostats:
1060 10    print "Scheduling thermostat {0} for zone {1}".format(thermostat["?tstat"], thermostat["?zone"])
1061 11    if time.after("18:00"): # 6pm, start nighttime schedule
1062 12      POST(thermostat["?api"], {"heating_setpoint": 60, "cooling_setpoint": 85})
1063 13    if time.after("8:00"): # 8am, start daytime schedule
1064 14      POST(thermostat["?api"], {"heating_setpoint": 72, "cooling_setpoint": 76})
1065 15    time.sleep(60) # run every minute

```

Fig. 11. Python pseudocode for a simple thermostat controller that enacts the same schedule over all HVAC zones

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 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

```

1079 1 zones = hodb.run_query("SELECT ?zone WHERE { ?zone rdf:type brick:HVAC_Zone }")
1080 2 for zone in zones:
1081 3     query = """SELECT ?uuid ?room ?tstat_api WHERE {
1082 4         ?room bf:isPartOf {0} . ?room bf:isLocationOf ?sensor .
1083 5         ?sensor rdf:type/rdfs:subClassOf* brick:Occupancy_Sensor . ?sensor bf:uuid ?uuid .
1084 6         {0} bf:isFedBy+/bf:isControlledBy ?tstat .
1085 7         ?tstat rdf:type brick:Thermostat . ?tstat bf:uri ?tstat_api .
1086 8     }""".format(zone["?zone"])
1087 9     tstat_config = hodb.run_query(query)
1088 10 # construct MDAL query with the following variable definition
1089 11 mdalquery["Variables"] = [{
1090 12     "Name": "occ",
1091 13     "Definition": query,
1092 14 }]
1093 15 occupancy_data = mdal.run_query(mdalquery)
1094 16 model = train_model(occupancy_data)
1095 17 schedule = schedule_from_model(model)
1096 18 execute_schedule(tstat_config["?tstat_api"], schedule)

```

Fig. 12. Python pseudocode for a thermostat controller that learns its schedule from a model trained on occupancy data from the rooms conditioned by the thermostat

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 12 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.

6.3 Discussion

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.

7 CONCLUSION

This paper has grappled with practical issues involved in integrating Brick metadata into real-world, latency-sensitive applications.

First, we characterize the graphs and queries that constitute the Brick workload. We find 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.

Second, we present a performance evaluation of current, popular RDF databases against the Brick workload, and demonstrate that none of them meet the latency target of 100ms.

Next, we use our 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.

8 ACKNOWLEDGEMENTS

This research is sponsored in part by National Science Foundation CPS-1239552.

REFERENCES

- [1] 1999. RDF Concepts Namespace. <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. (1999).
- [2] 2000. RDF Schema Namespace. <https://www.w3.org/2000/01/rdf-schema#>. (2000).
- [3] 2017. The Go Programming Language. <https://golang.org/>. (2017).
- [4] Jans Aasman. 2006. Allegro graph: RDF triple database. *Cidade: Oakland Franz Incorporated* (2006).
- [5] Michael P Andersen and David E Culler. 2016. BTRDB: optimizing storage system design for timeseries processing. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)*.
- [6] Grigoris Antoniou and Frank Van Harmelen. 2004. Web ontology language: Owl. In *Handbook on ontologies*. Springer, 67–92.
- [7] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, et al. 2016. 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.
- [8] David Beckett, T Berners-Lee, E Prud'hommeaux, and G Carothers. 2014. RDF 1.1 Turtle. *World Wide Web Consortium* (2014).
- [9] Arka Bhattacharya, Joern Ploennigs, and David Culler. 2015. 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, 33–34.
- [10] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. 2009. DBpedia-A crystallization point for the Web of Data. *Web Semantics: science, services and agents on the world wide web* 7, 3 (2009), 154–165.
- [11] Christian Bizer and Andreas Schultz. 2009. The berlin sparql benchmark. <http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/>. (2009).
- [12] Cayleygraph. 2017. Cayley. <https://cayley.io>. (2017).
- [13] Leonidas Deligiannidis, Krys J Kochut, and Amit P Sheth. 2007. RDF data exploration and visualization. In *Proceedings of the ACM first workshop on CyberInfrastructure: information management in eScience*. ACM, 39–46.
- [14] Inc Dgraph Labs. 2017. Dgraph. <https://dgraph.io/index.html>. (2017).
- [15] Orri Erling and Ivan Mikhailov. 2009. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*. Springer, 7–24.
- [16] Gabriel Fierro and David E. Culler. 2017. Design and Analysis of a Query Processor for Brick. In *Proceedings of the ACM International Conference on Embedded Systems for Energy-Efficient Built Environments (BuildSys)*. ACM.
- [17] Inc Franz. 2017. AllegroGraph: Semantic Graph Database. <https://allegrograph.com/allegrograph/>. (2017).
- [18] Flavius Frasincar, Alexandru Telea, and Geert-Jan Houben. 2006. Adapting graph visualization techniques for the visualization of RDF data. *Visualizing the semantic web* 2006 (2006), 154–171.
- [19] Sadyuki Furuhashi. 2017. MessagePack: It's like JSON. but fast and small, 2014. URL <http://msgpack.org> (2017).
- [20] Inc Google. 2017. Badwolf. <https://google.github.io/badwolf/>. (2017).
- [21] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 query language. *W3C recommendation* 21, 10 (2013).
- [22] Andreas Harth and Stefan Decker. 2005. Optimized index structures for querying rdf from the web. In *Web Congress, 2005. LA-WEB 2005. Third Latin American*. IEEE, 10–pp.
- [23] Philipp Heim, Sebastian Hellmann, Jens Lehmann, Steffen Lohmann, and Timo Stegemann. 2009. RelFinder: Revealing Relationships in RDF Knowledge Bases. *SAMT* 5887 (2009), 182–187.
- [24] Ora Lassila and Ralph R Swick. 1999. Resource description framework (RDF) model and syntax specification. (1999).
- [25] Kamesh Madduri and Kesheng Wu. 2011. Massive-scale RDF processing using compressed bitmap indexes. In *International Conference on Scientific and Statistical Database Management*. Springer, 470–479.
- [26] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.
- [27] George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41.

- [28] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. 2011. DBpedia SPARQL benchmark-performance assessment with real queries on real data. *The Semantic Web-ISWC 2011* (2011), 454–469.
- [29] Inc Neo Technology. 2017. Neo4j. <https://neo4j.com/>. (2017).
- [30] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment* 1, 1 (2008), 647–659.
- [31] Jakob Nielsen. 1994. *Usability engineering*. Elsevier.
- [32] Natalya F Noy, Michael Sintek, Stefan Decker, Monica Crubézy, Ray W Ferguson, and Mark A Musen. 2001. Creating semantic web contents with protege-2000. *IEEE intelligent systems* 16, 2 (2001), 60–71.
- [33] Harshal Patni, Cory Henson, and Amit Sheth. 2010. Linked sensor data. In *Collaborative Technologies and Systems (CTS), 2010 International Symposium on*. IEEE, 362–370.
- [34] Mary Ann Piette, Sila Kiliccote, and Girish Ghatikar. 2014. Field experience with and potential for multi-time scale grid transactions from responsive commercial buildings. (2014).
- [35] Eric Prud, Andy Seaborne, et al. 2006. SPARQL query language for RDF. (2006).
- [36] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. ACM, 1–10.
- [37] Craig Sayers. 2004. Node-centric rdf graph visualization. *Mobile and Media Systems Laboratory, HP Labs* (2004).
- [38] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. 2009. SP2Bench: a SPARQL performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 222–233.
- [39] OpenLink Software. 2017. Virtuoso. <https://virtuoso.openlinksw.com/download/>. (2017).
- [40] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. 2008. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*. ACM, 595–604.
- [41] SYSTAP, LLC. 2017. Bigdata Database Architecture Whitepaper. https://www.blazegraph.com/whitepapers/bigdata_architecture_whitepaper.pdf. (2017).
- [42] SYSTAP, LLC. 2017. blazegraph. <https://www.blazegraph.com/>. (2017).
- [43] The Apache Software Foundation. 2017. A free and open source Java framework for building Semantic Web and Linked Data applications. <https://jena.apache.org/> (2017).
- [44] The Apache Software Foundation. 2017. High performance Triple Datastore. <https://jena.apache.org/documentation/tdb/>. (2017).
- [45] The RDFLib Team. 2017. RDFLib. <https://rdflib.readthedocs.io/en/stable/>. (2017).
- [46] TopQuadrant. 2017. TopBraid Live. <http://www.topquadrant.com/products/topbraid-live/>. (2017).
- [47] Kesheng Wu, Sean Ahern, E Wes Bethel, Jacqueline Chen, Hank Childs, Estelle Cormier-Michel, Cameron Geddes, Junmin Gu, Hans Hagen, Bernd Hamann, et al. 2009. FastBit: interactively searching massive data. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012053.