Design and Analysis of a Query Processor for Brick

Gabe Fierro
UC Berkeley
gtfierro@cs.berkeley.edu

David E. Culler
UC Berkeley
culler@cs.berkeley.edu

ABSTRACT

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.

CCS CONCEPTS

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

KEYWORDS

Smart Buildings, Building Management, Metadata, Graph Database, RDF, SPARQL

ACM Reference format:
Gabe Fierro and David E. Culler. 2017. Design and Analysis of a Query Processor for Brick. In *Proceedings of BuildSys '17, Delft, Netherlands, November 8–9, 2017,* 10 pages.
https://doi.org/10.1145/3137133.3137155

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 [8]. This set of requirements drove the development of Brick [6], a recently proposed metadata standard for describing the set of entities and relationships within a building. Brick succeeds along three metrics: 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 [19] and SPARQL query language [28] for representing and querying graphs, respectively. Initial work [6] 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 [25]. 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.

BuildSys '17, November 8–9, 2017, Delft, Netherlands

ACM ISBN 978-1-4503-5544-5/17/11…$15.00
https://doi.org/10.1145/3137133.3137155

2.1 Brick Overview

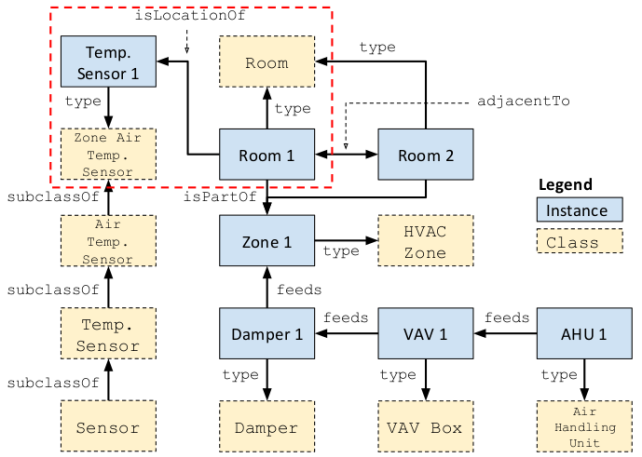


Figure 1: Example Brick graph for a simple building, showing class instantiations and subclass declarations. Solid blue boxes make up the building, and dotted tan boxes represent the class hierarchy. Note the long chains of `feeds` and `subClassOf` relationships.

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 a 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 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.

```
### VAV Enum (Building Dashboard)
SELECT DISTINCT ?vav WHERE {
  ?vav rdf:type brick:VAV .
}
### Temp Sensors (Building Dashboard, Room Diagnostics)
SELECT DISTINCT ?sensor WHERE {
  ?sensor rdf:type/rdfs:subClassOf* brick:Zone_Temperature_Sensor .
}
### AHU Children (Building Dashboard)
SELECT DISTINCT ?x WHERE {
  ?ahu rdf:type brick:AHU .
  ?ahu bf:feeds+ ?x .
}
### Spatial Mapping (Building Dashboard)
SELECT DISTINCT ?floor ?room ?zone WHERE {
  ?floor rdf:type brick:Floor .
  ?room rdf:type brick:Room .
  ?zone rdf:type brick:HVAC_Zone .
  ?room bf:isPartOf+ ?floor .
  ?room bf:isPartOf+ ?zone .
}
### Sensors In Rooms (Room Diagnostics)
SELECT DISTINCT ?sensor ?room
WHERE {
  { ?sensor rdf:type/rdfs:subClassOf* brick:Zone_Temperature_Sensor . }
  UNION
  { ?sensor rdf:type/rdfs:subClassOf* brick:Discharge_Air_Temperature_Sensor . }
  UNION
  { ?sensor rdf:type/rdfs:subClassOf* brick:Occupancy_Sensor . }
  UNION
  { ?sensor rdf:type/rdfs:subClassOf* brick:CO2_Sensor . }
  ?vav rdf:type brick:VAV .
  ?zone rdf:type brick:HVAC_Zone .
  ?room rdf:type brick:Room .
  ?vav bf:feeds+ ?zone .
  ?zone bf:hasPart ?room .
  {?sensor bf:isPointOf ?vav }
  UNION
  {?sensor bf:isPointOf ?room }
}
### VAV Relships (Building Dashboard)
SELECT DISTINCT ?vav ?x ?y ?z ?a ?b WHERE {
  ?vav rdf:type brick:VAV .
  ?vav bf:feeds+ ?x .
  ?vav bf:isFedBy+ ?y .
  ?vav bf:hasPoint+ ?z .
  ?vav bf:hasPart+ ?a .
}
### Grey Box (Automatic Grey Box Modeler)
SELECT DISTINCT ?vav ?room ?temp_uuid ?valve_uuid ?setpoint_uuid WHERE {
  ?vav rdf:type brick:VAV .
  ?vav bf:hasPoint ?tempsensor .
  ?tempsensor rdf:type/rdfs:subClassOf* brick:Temperature_Sensor .
  ?tempsensor bf:uuid ?temp_uuid .
  ?vav bf:hasPoint ?valvesensor .
  ?valvesensor rdf:type/rdfs:subClassOf* brick:Valve_Command .
  ?valvesensor bf:uuid ?valve_uuid .
  ?vav bf:hasPoint ?setpoint .
  ?setpoint rdf:type/rdfs:subClassOf* brick:Zone_Temperature_Setpoint .
  ?setpoint bf:uuid ?setpoint_uuid .
  ?room rdf:type brick:Room .
  ?tempsensor bf:isLocatedIn ?room .
}
```

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

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 subsystems. 100ms is a common target for users to feel an interaction is "instantaneous" [25].

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

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

Figure 3: The set of triples for the highlighted part of the graph in Figure 1, using the Turtle format for RDF data [7].

metadata model should be minimal to leave more time for the rest of the computation.

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

Figure 2 shows the queries constituting these applications. Other categories of applications that can benefit from fast metadata queries are fast demand response [27], model-predictive control, and online fault detection and diagnosis. [6] and [8] present more comprehensive lists of metadata-driven applications.

2.3 RDF Data Model

Brick graphs are specified using the RDF data model [19]. 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. The RDF data model can also represent literal values, which Brick uses to store information such as coordinates and pointers to timeseries streams.

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) [28] 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 [17] 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, 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.

`TempSensors` finds all sensors that are instances of zone temperature sensors or any subclass thereof.

`AHUChildren` lists all equipment and sensors downstream of an air handler unit.

`SpatialMapping` associates floors, the rooms on that floor, and the HVAC zones that cover those rooms.

`SensorsInRooms` associates a family of sensors with a room, using the room's HVAC zone and VAV information.

`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.

`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.

3.1 RDF Databases

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

Apache Jena [35] 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 [18] (explained in §4), which is a common implementation approach.

Blazegraph [33, 34] 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

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

Table 1: Query latency distribution for the small building (CIEE). All times are in milliseconds. A – denotes the query did not return any results. Bold indicates that the 99th percentile latency is outside the 100ms bound.

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

Table 2: Query latency distribution for a large building (Soda).

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

Table 3: Query latency distribution for a large building (SDH).

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 [24] 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 6.

RDFLib [37] 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.

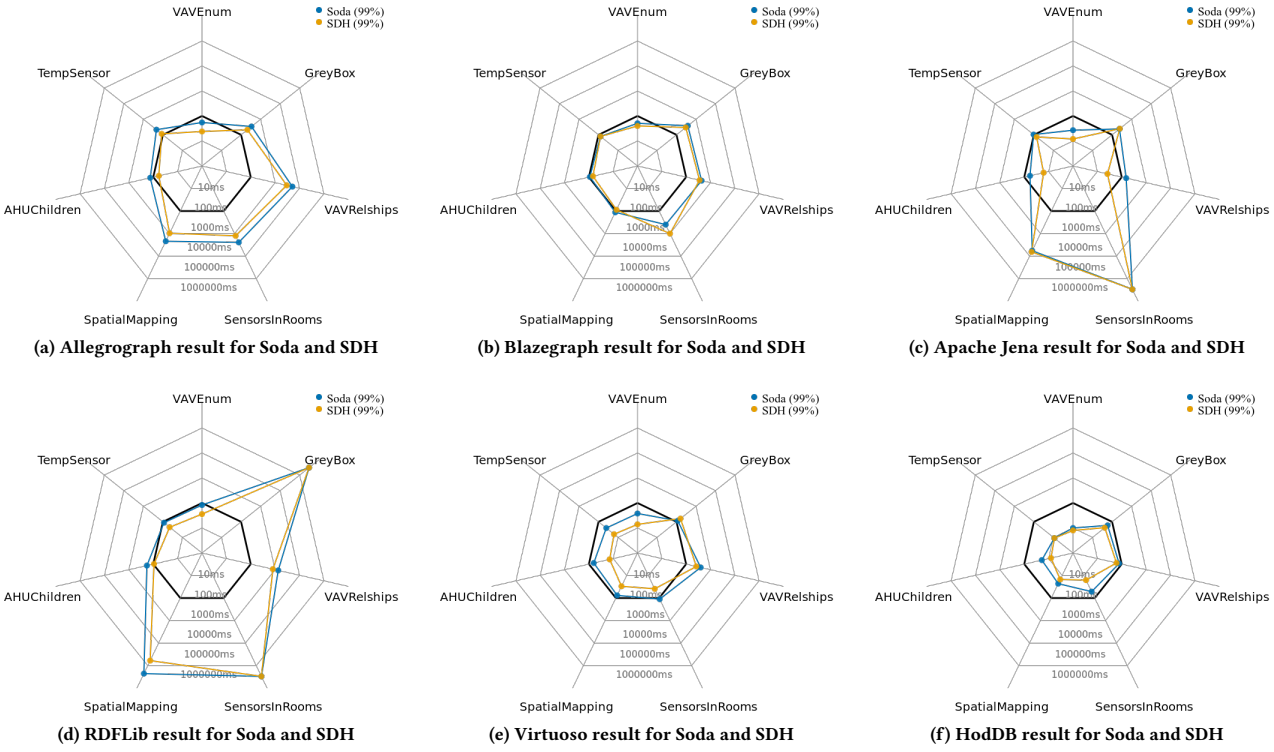
Allegrograph [4, 14] is an ACID-compliant, commercial, closed-source graph database for storing billions of RDF triples. It provides a full SPARQL 1.1 implementation in addition to support for geospatial and temporal data.

Virtuoso [13, 31] 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 [38], for which we could not obtain a copy, and the RDF extension [20] to the FastBit [39] 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 [11], Dgraph [12], Badwolf [16] and Neo4j [23]), which implement alternative graph query languages such as Gremlin [29] 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



(a) Allegrograph result for Soda and SDH

(b) Blazegraph result for Soda and SDH

(c) Apache Jena result for Soda and SDH

(d) RDFLib result for Soda and SDH

(e) Virtuoso result for Soda and SDH

(f) HodDB result for Soda and SDH

Figure 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.

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 5 (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.

We develop a simple Python test harness[1] to dispatch each benchmark query against each database and measure the time in milliseconds from the time the query was dispatched to the time 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 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 [5]. Most of the RDF databases we tested do not implement the necessary inference, so each Brick graph had to be pre-populated with the set of all inverse edges because the queries were not written with knowledge of which of the inverse edges were used in the original definition of the building.

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

3.3 Brick Query Performance

Tables 1, 2 and 3 show the mean, standard deviation and 99th percentile latencies for each of the benchmark queries (Figure 2) over the three Brick buildings from Table 5. 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 99^{th} percentile results to draw attention to how well each database meets the performance target (the bold heptagon).

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

Most databases exhibit good query performance (within the 100ms bound) on the small building (Table 1), but substantially degraded performance on the two larger buildings (Tables 2 and 3). Only Allegrograph, Blazegraph and Virtuoso are able to complete each query on the two large Brick buildings in less than 5 minutes[2]. Virtuoso performs closest to the 100ms latency target: its 99th percentile latency fails only on `VAVRelships` and `GreyBox`.

To understand the demands the Brick workload places on a query processor, we examine which query features exhibit poor performance across buildings and databases. Over the suite of queries in Table 6, 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 [10, 22]. 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 4 shows the mean, standard deviation and 99th percentile of the resulting query latencies. Allegrograph, Blazegraph, RDFLib and Virtuoso all exhibit a dramatic 200-300% increase in execution time when the query pattern contains the * or + operators.

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 rather than relying on optimized joins between its B-tree indices.

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

Now that we 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 5 compares the size and density of several real-world datasets (DBPedia Infobox [9], LinkedSensor [26] and Wordnet [21]), synthetic datasets (BSBM [10] and SP2B [30]) and Brick models. We draw several conclusions: firstly, Brick graphs are a few orders of magnitude smaller (in number of triples and nodes) and tend not to use as many edge types as other RDF datasets. Secondly, for each edge type, Brick graphs have a higher average fanout. This increases the size of range queries over YARS-style B-tree indices, which can cause a drop in performance.

4.2 SPARQL Features

Brick queries only require a subset of features defined by the SPARQL 1.1 specification [17]. 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 6 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

```
?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.

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

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

Table 4: Effect of property path operators on query execution time. All times are in milliseconds. Uses Soda Brick model.

	RDF Dataset	Triples	Nodes	Edge Types	Avg Out Degree per Edge Type
Real	Infobox [9]	30,024,092	9,741,482	2063	.0015
	Wordnet [21]	8,574,806	2,487,208	64	.0539
	Sensor [26]	185,950	86,580	12	.179
Syn.	SP2B [30]	7,442	4,800	57	.0272
	BSBM [10]	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 5: Graph properties of some published RDF datasets and three representative Brick models.

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

Table 6: Properties of the benchmark SPARQL queries

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 [18], uses a collection of B-tree indices to index the dataset by all triples, pairs and single values that could be involved in a query. Each node and edge (**s**ubject, **p**redicate and **o**bject) is assigned a short, unique identifier. Each triple is rewritten using these IDs before being arranged and inserted into six covering indices: SPO, SOP, OSP, OPS, PSO, POS. The indices make use of fast B-tree range traversal to enumerate matching triples; for example, the SPARQL term `?ahu rdf:type brick:AHU` could find all matching subjects by traversing the POS index and looking for all entries with a PO prefix matching the concatenation of `rdf:type` and `brick:AHU`. YARS [18], RDFLib [37], RDF3X [24], Blazegraph [33] and the TDB engine behind Jena [36] 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. 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.

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 [18]. 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 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

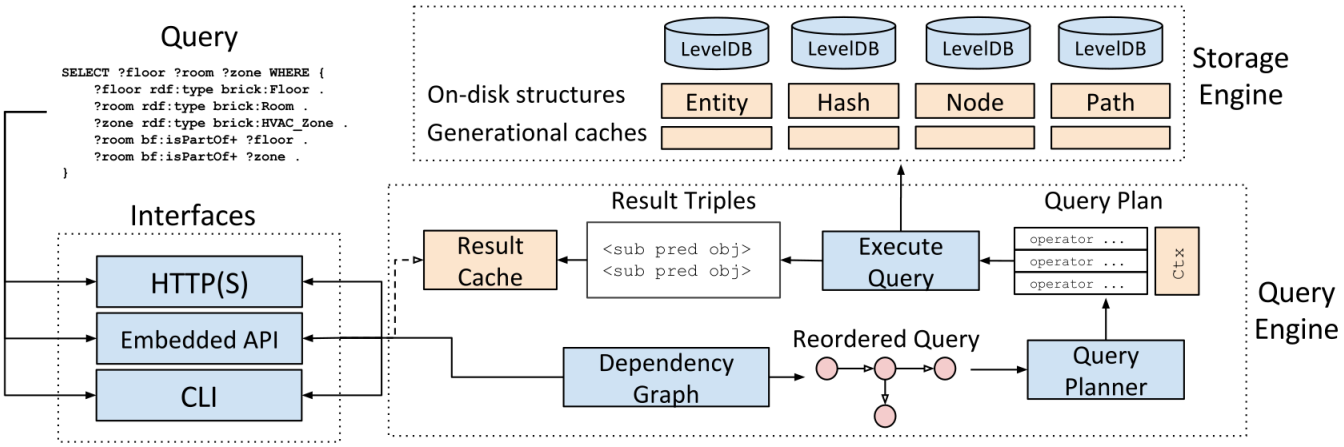


Figure 5: Architecture of HodDB

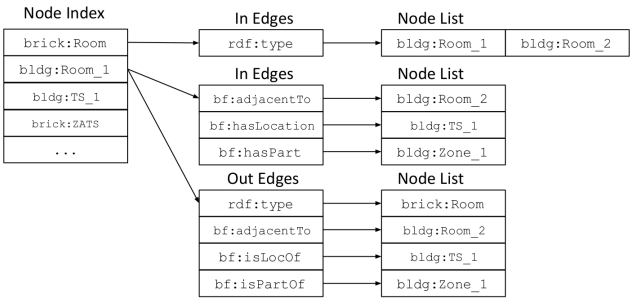


Figure 6: Node index structure for part of Figure 1.

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 LevelDB [3], 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 [15]-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 5), iterating through a B-tree index can result in worse performance

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

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 1, 2 and 3). The **predicate index** is similar in structure to the node index, but uses predicate/edge hashes as keys.

Path Index: The path index accelerates evaluation of queries involving chains of 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 briefly 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. 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 [32]. 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 turns the dependency graph into a flat list of graph operators. HodDB defines an operator for each possible combination of variables and entities in a query. An operator is a small piece of code that takes a SPARQL pattern and a query context (described below) 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 set of operators output from the query planner, using a *context* object to store all intermediate state. The context object stores a set of candidate entities for each variable. For each of these entities, the context object associates variable names to sets of candidate entities linked to the original entity through one or more SPARQL patterns.

All candidate sets of entities are stored as in-memory B-trees, which double as the join structure. The advantage of this approach is when an operator pulls a set of entities out of the node or path index, the values in the query context do not require any preprocessing for the join. Once all operators have been executed, HodDB iterates through the chains of candidate entities to extract sequences of entities 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.

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 <4ms on a cache hit.

Implementation: HodDB is free and open-source [5] 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 4, HodDB's path index means that property path operators only induce a 38% overhead on query execution time. Table 7 compares disk usage for 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.

Brick Workload: We now refer back to Tables 1, 2 and 3; the last column shows the query latency distribution for HodDB. The mean latencies are all below 50ms, and the 99th percentile latencies (influenced mostly by garbage collection pauses) are all below the performance target of 100ms.

This performance is possible because HodDB targets a small, well-defined domain within the RDF/SPARQL space. An obvious

[4] Which is how MySQL's optional result cache works
[5] https://hoddb.org/
[6] Go version 1.8.1 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 7: Disk space usage for each graph. HodDB's indices are small — about the same size as RDF-3X's compressed B-trees.

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. The two large buildings used in the evaluation are representative of Brick model size and complexity, but initial experiments suggest that HodDB will perform well for models of up to 100k triples.

6 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.

Lastly, 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.

7 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] Grigoris Antoniou and Frank Van Harmelen. 2004. Web ontology language: Owl. In *Handbook on ontologies*. Springer, 67–92.
[6] 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.*
[7] David Beckett, T Berners-Lee, E Prud'hommeaux, and G Carothers. 2014. RDF 1.1 Turtle. *World Wide Web Consortium* (2014).
[8] 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.
[9] 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.
[10] Christian Bizer and Andreas Schultz. 2009. The berlin sparql benchmark. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/. (2009).
[11] Cayleygraph. 2017. Cayley. https://cayley.io. (2017).
[12] Inc Dgraph Labs. 2017. Dgraph. https://dgraph.io/index.html. (2017).
[13] Orri Erling and Ivan Mikhailov. 2009. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*. Springer, 7–24.
[14] Inc Franz. 2017. AllegroGraph: Semantic Graph Database. https://allegrograph.com/allegrograph/. (2017).
[15] Sadayuki Furuhashi. 2017. MessagePack: It's like JSON. but fast and small, 2014. *URL http://msgpack. org* (2017).
[16] Inc Google. 2017. Badwolf. https://google.github.io/badwolf/. (2017).
[17] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 query language. *W3C recommendation* 21, 10 (2013).
[18] 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.
[19] Ora Lassila and Ralph R Swick. 1999. Resource description framework (RDF) model and syntax specification. (1999).
[20] 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.
[21] George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41.
[22] 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.
[23] Inc Neo Technology. 2017. Neo4j. https://neo4j.com/. (2017).
[24] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment* 1, 1 (2008), 647–659.
[25] Jakob Nielsen. 1994. *Usability engineering*. Elsevier.
[26] Harshal Patni, Cory Henson, and Amit Sheth. 2010. Linked sensor data. In *Collaborative Technologies and Systems (CTS), 2010 International Symposium on*. IEEE, 362–370.
[27] 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).
[28] Eric Prud, Andy Seaborne, et al. 2006. SPARQL query language for RDF. (2006).
[29] 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.
[30] 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.
[31] OpenLink Software. 2017. Virtuoso. https://virtuoso.openlinksw.com/download/. (2017).
[32] 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.
[33] SYSTAP, LLC. 2017. Bigdata Database Architecture Whitepaper. https://www.blazegraph.com/whitepapers/bigdata_architecture_whitepaper.pdf. (2017).
[34] SYSTAP, LLC. 2017. blazegraph. https://www.blazegraph.com/. (2017).
[35] 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).
[36] The Apache Software Foundation. 2017. High performance Triple Datastore. https://jena.apache.org/documentation/tdb/. (2017).
[37] The RDFLib Team. 2017. RDFLib. https://rdflib.readthedocs.io/en/stable/. (2017).
[38] TopQuadrant. 2017. TopBraid Live. http://www.topquadrant.com/products/topbraid-live/. (2017).
[39] 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.