NoSQL Databases for RDF: An Empirical Evaluation

P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. L. Keppmann, D. Miranker, J. F. Sequeda, M. Wylot

Presented by: Besat Kassaie

Outline

- Objectives
- Evaluated Systems
- Experiments and Results
- Conclusion
- Q&A

Objectives

- Comparing NoSQL systems with native triple stores
- Finding performance similarities between systems
- Providing an environment for replicable tests
  - (paper’s website is not available anymore!)
- Not choosing a **“winner”** among systems

Outline

- Objectives
- Evaluated Systems
- Experiments and Results
- Conclusion
- Q&A

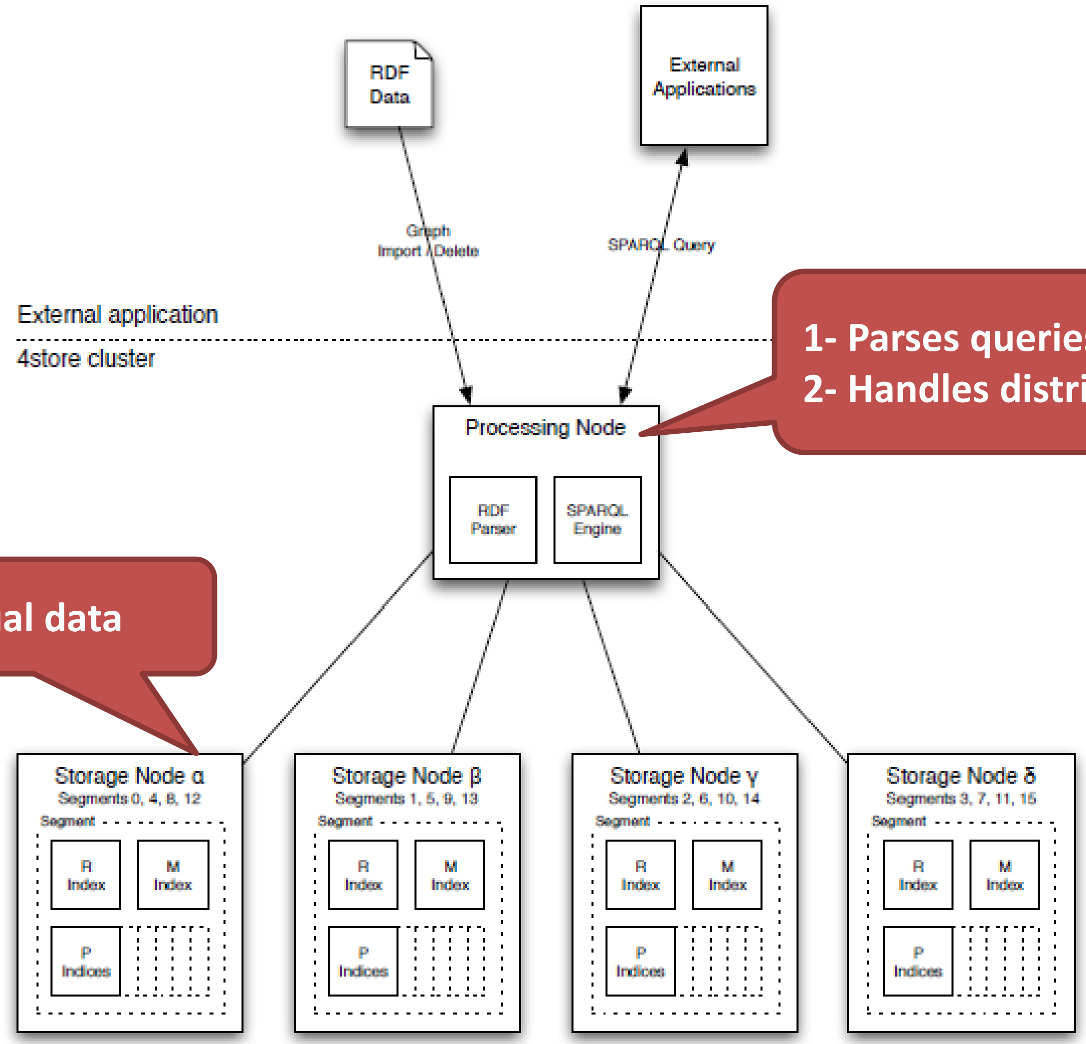
Evaluated Systems

Systems selected based on two factors:

- Current extensions on NoSQL for supporting RDF
- Covering different NoSQL system types

Storage System	Type
CouchDB	Document Based
Cassandra	Key-Value/Column store
HBase	Key-Value/Column store
4store (Baseline)	Distributed RDF DBMS

4store Architecture



1- Parses queries

2- Handles distributed communication

Stores actual data

Figure taken from [1]

4store

- RDF data stored as quads:
  - (model, subject, predicate, object)
- Encodes URIs, literals and blank nodes as numbers
- Keeps data in property tables
- Divides data among non-overlapping segments based on "subject"

segment = (subject code) mod segments

4store Indices

- P indices
  - Two P indices per predicate :
    - Based on Subject(s p ?) → Find Objects for given PS
    - Based on Object (? p o)→ Find Subjects for given PO
    - (? ? o) or (s ? ?) → Search all P indices
- R index
  - Maps encoded hash value of P,O,S → String
- M index
  - For a given model → List of all triples

HBase

- Column-oriented NoSQL
- Columns grouped into Column Families
- Data Sorted lexicographically by row-key
- Multi-dimensional: row, column, timestamp
- Relies on HDFS
- Integrated by Hadoop (MapReduce)

Jena+HBase

Schema : Leverages sorted row-keys in HBase

- Maps literals, URIs → 8-byte ids
- RDF data is stored in three index tables
  - SPO, POS and OSP

Row-Key	ColName→ Empty
SPO1	Empty
SPO2	Empty
…….	……..

(s ? ?) , (s p ?)

Row-Key	ColName→ Empty
POS1	Empty
POS2	Empty
…….	……..

(? p o) , (? p ?)

Row-Key	ColName→ Empty
OSP1	Empty
OSP2	Empty
…….	……..

(? ? o) , (s ? o)

Jena+HBase

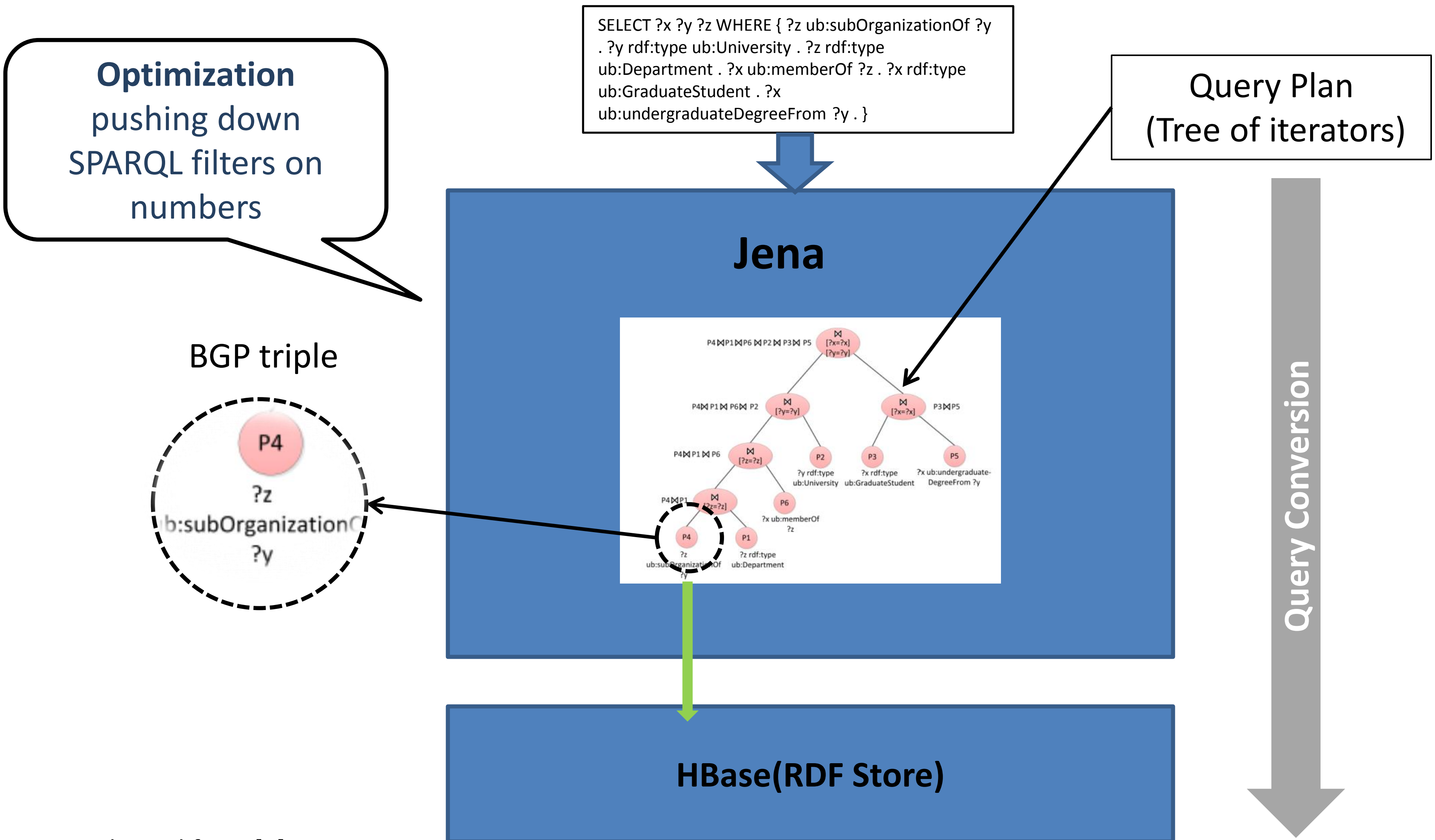


Figure adopted from [2]

Hive+HBase

Schema :

- Compressed subjects are row-keys
- TimeStamps used to store multi-valued objects

Row-Key	TimeStamp	Predicate1	Predicate2
Compressed Subject1	1	Object1	Object11
	2	Object4	Empty
	3	Object7	Empty
…….	………	……..	………

Hive+HBase

Query Conversion

1. Unique Subjects are identified in query's BGP by *Jena ARQ*
2. For each Subject a temporary Hive Table is created containing requested predicates and objects
3. Join conditions are identified to see which temp tables need to be joined
4. Hive queries for joins are executed by MapReduce

CumulusRDF

- RDF Store based on Cassandra+Sesame
- Cassandra indices:
  - Hash index based on row-key
  - Sorted index for column names(columns are sorted)
  - Secondary index mapping values to row-key

CumulusRDF

Schema: leverages Cassandra indices

Row-Key	ColName →PO1	ColName →PO2
S1	Empty	Empty
S2	Empty	Empty
…….	……..	

(s ? ?) , (s p ?)

Row-Key	ColName → SP1	ColName → SP2
O1	Empty	Empty
O2	Empty	Empty
…….	……..	……..

(? ? o) , (s ? o)

Row-Key	ColName → S1	ColName → S2
PO1	Empty	Empty
PO2	Empty	Empty
…….	……..	……..

(? p o)

Row-Key	ColName → P1	ColName → P2
PO1	P1	Empty
PO2	Empty	p2
…….	……..	……..

(? p ?) *(not used here)*

CumulusRDF

Query

- “Sesame is a powerful Java framework for processing and handling RDF data”[3]
- Sesame translates SPARQL queries to index lookups on Cassandra indices
- Sesame processes joins and filters

Couchbase

- NoSQL document-oriented database
- Supports JSON documents

Schema

- RDF data is serialized to JSON Documents
- Subjects are document IDs
- Two JSON arrays for predicates and objects in each document

Couchbase

Query

- Query execution is based on Jena SPARQL engine (similar to HBase)
- Three Couchbase views are built to cover

  (? p ?), (? ? o), (? p o)
- For patterns including subject the entire JSON document is retrieved and parsed

Outline

- Objectives
- Evaluated Systems
- Experiments and Results
- Conclusion
- Q&A

Experimental Setting

- Benchmarks:
  - Berlin SPARQL Benchmark(BSBM)
    - ~ 10 million triples (Scale Factor: 28,850)
    - ~ 100 million triples(Scale Factor: 284,826)
    - ~ 1 billion triples(Scale Factor: 2,878,260)
  - DBpedia SPARQL Benchmark(DBPSB)
    - 153,737,783 triples (Scale Factor: 100%)

Results <4store>

- Query time decreases as #nodes increases
- Response time is sub-second for 10-100M triples
- 4store is slow for queries touching a lot of data (Q5)
- 4store times out for loading 1 billion triples

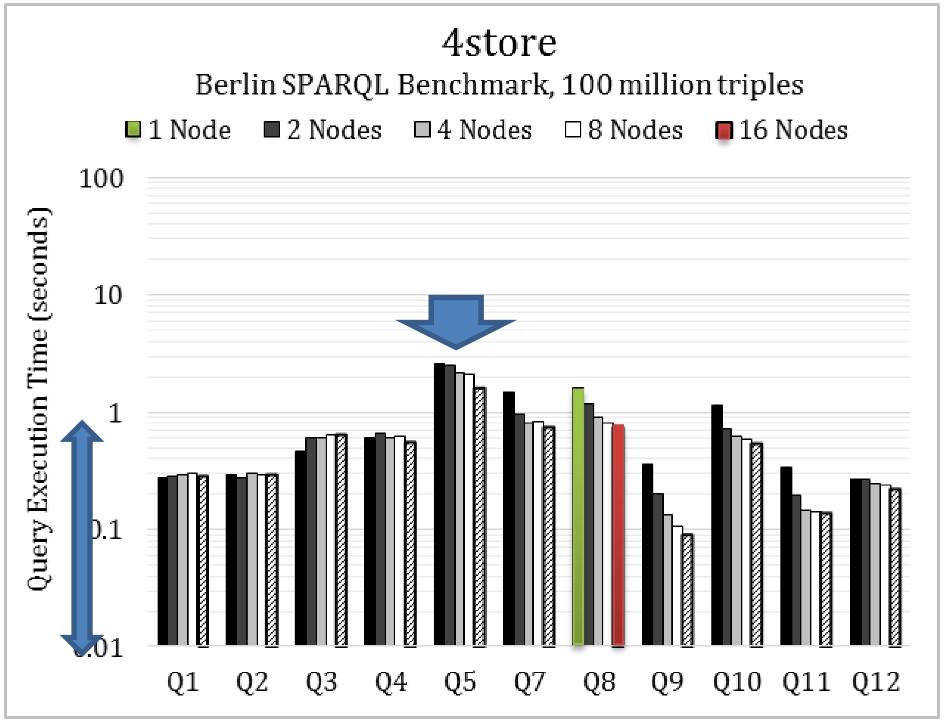
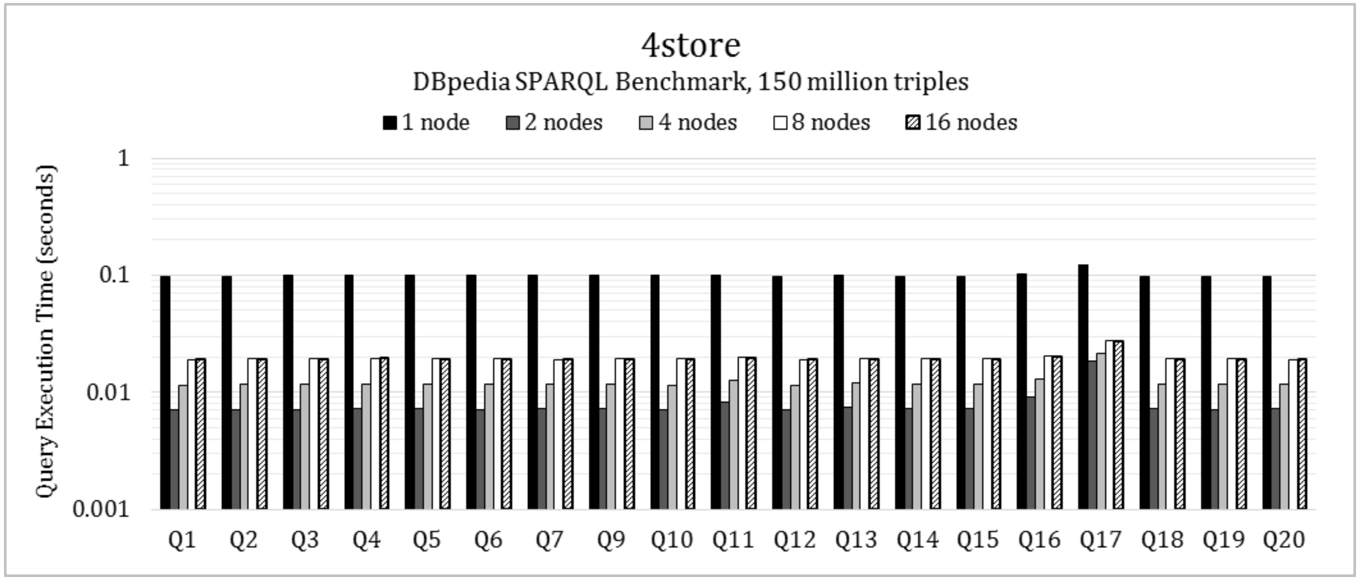


Figure taken from [4]

Results <4store>



- 4store is not scalable for DBpedia benchmark

highly complex dataset → too much fragmentation→ high network delays

Figure taken from [4]

Results <Jena+HBase>

- Sub-second query time for highly selective queries (Q2,Q8,Q9,Q11,Q12)
- System is slow for queries touching a lot of data (Q5,Q7)
- System times out for Q10 that has a filter on date

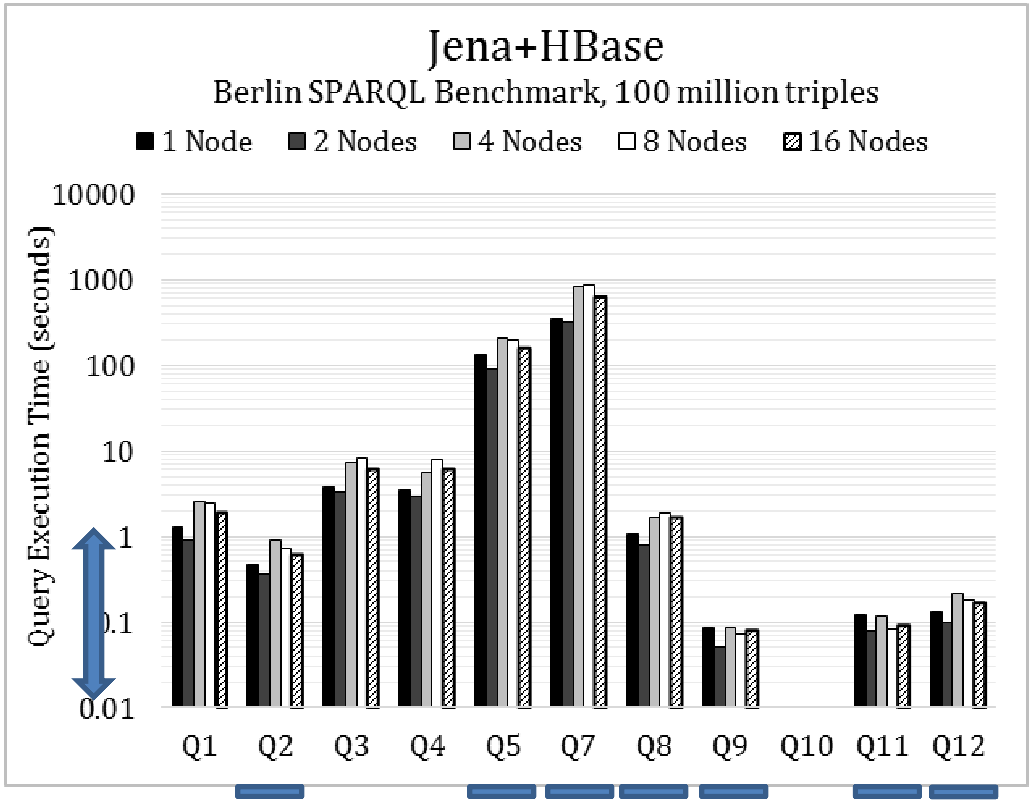
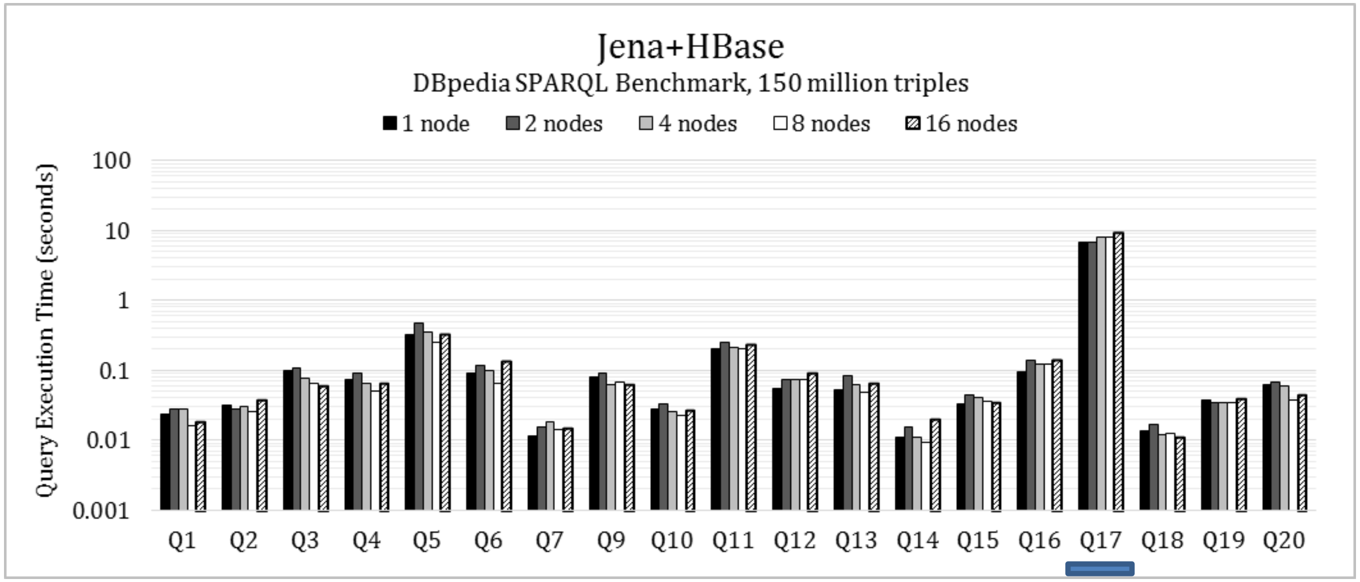


Figure taken from [4]

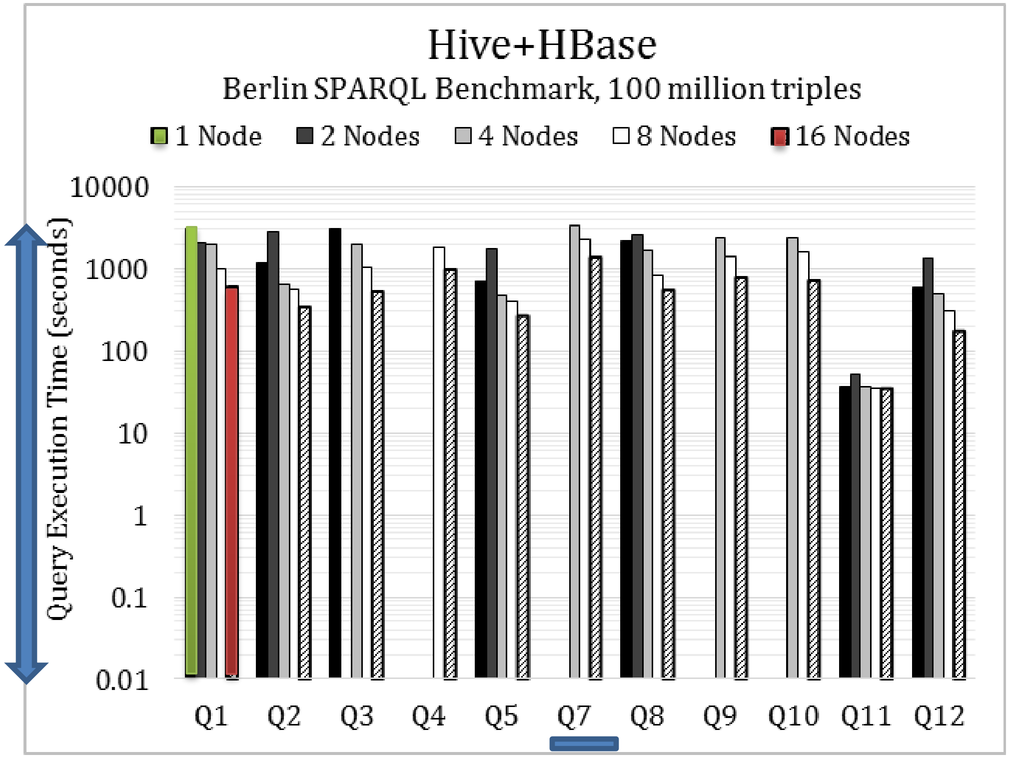
Results <Jena+HBase>



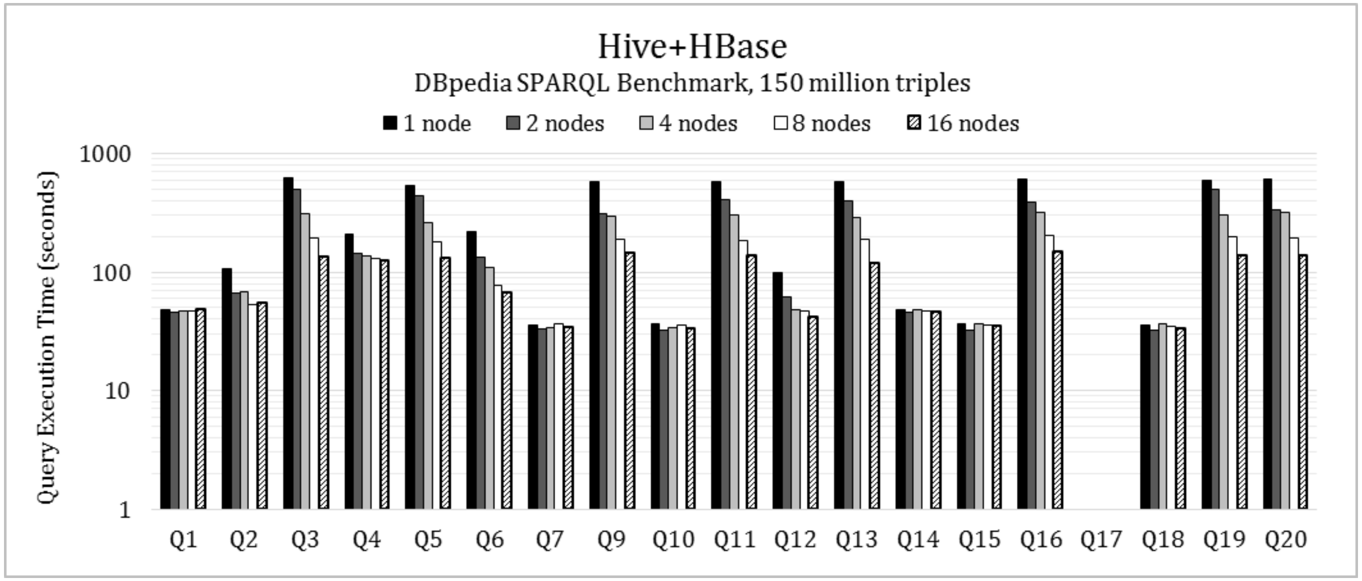
- Sub-second query time almost for all queries
  - Many duplicated rows are removed during loading
  - Queries are simpler than BSBM
- Q17 is slower due to filter on string
- Not scalable for this dataset

Result<Hive+HBase>

- Query time decreases as #nodes increase
- MapReduce shuffle stage increases query time (minute)
- Q7 is slowest (needs 4 MapReduce jobs)



Result<Hive+HBase>



- Scalable
- Low query time due to simpler queries with almost no join (comparing with BSBM)

Figure taken from [4]

Result<CulumusRDF>

- Performance decreases as # nodes and data size increase(heavy network communication)
- Q5 exceeds 1 hour (touching a lot of data)
- Q1,Q3,Q4,Q5 were challenging (complex queries)

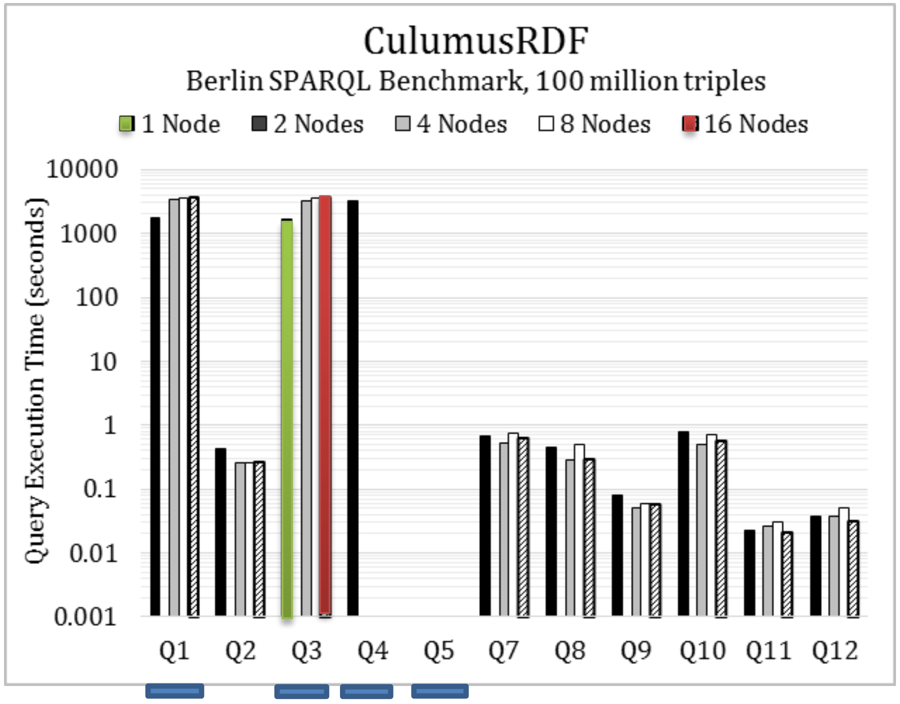
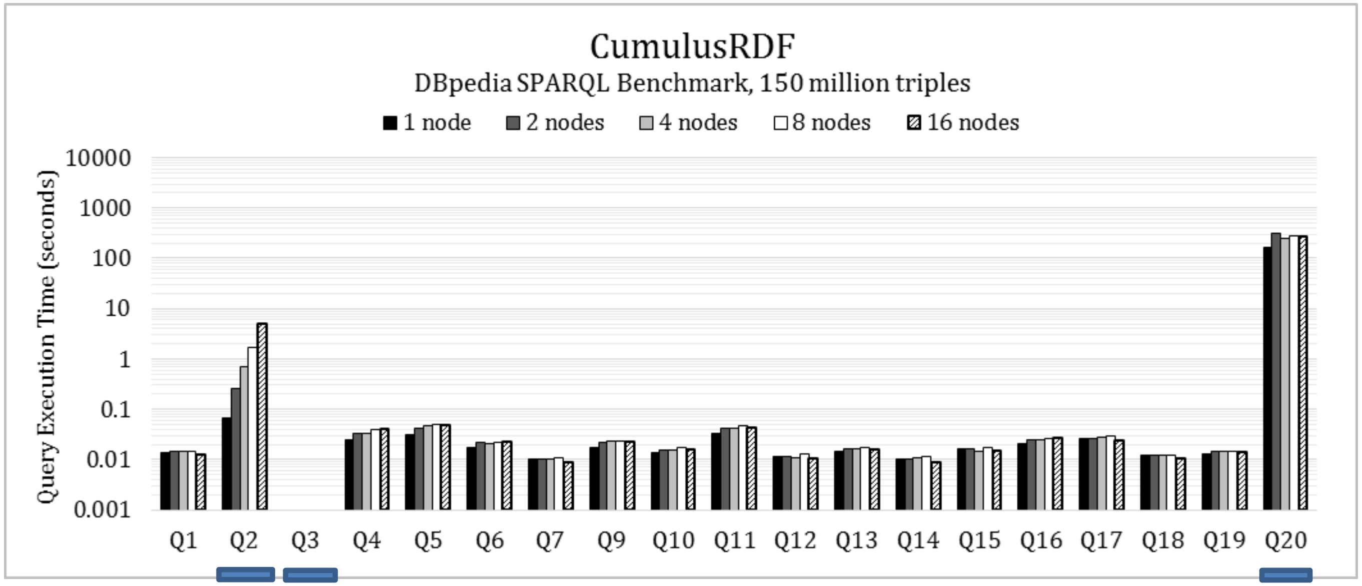


Figure taken from [4]

Result<CulumusRDF>



- System is very slow for Q2,Q3 and Q20 due to join on string
- Not scalable

Figure taken from [4]

Result<Couchbase>

- Encounters problem when loading 1 billion data to all cluster size
- System is fast for queries on 100M triples
- Q5 is slowest due to touching a lot of data
- DBpedia is similar

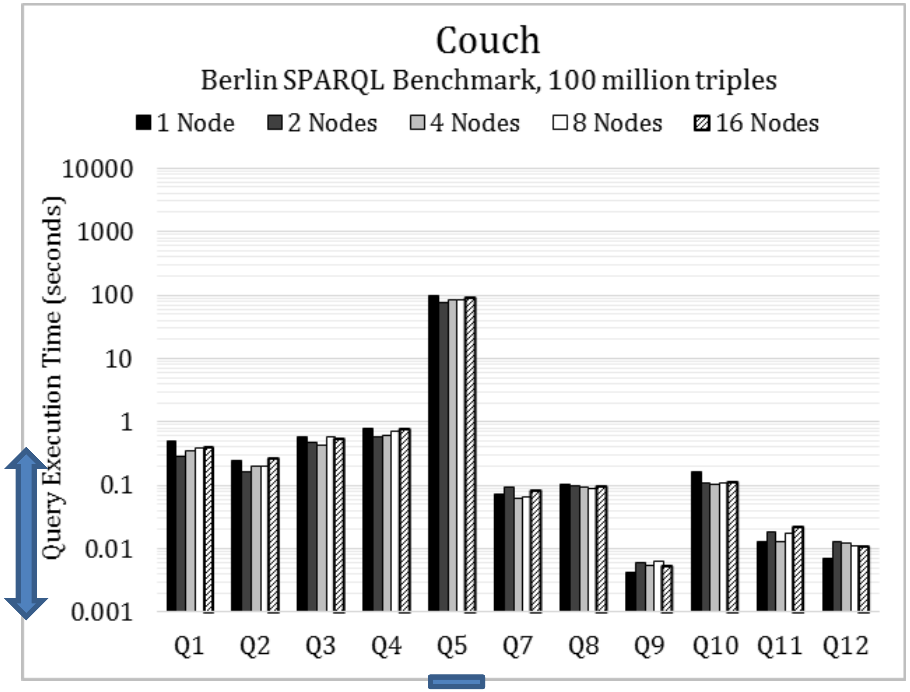
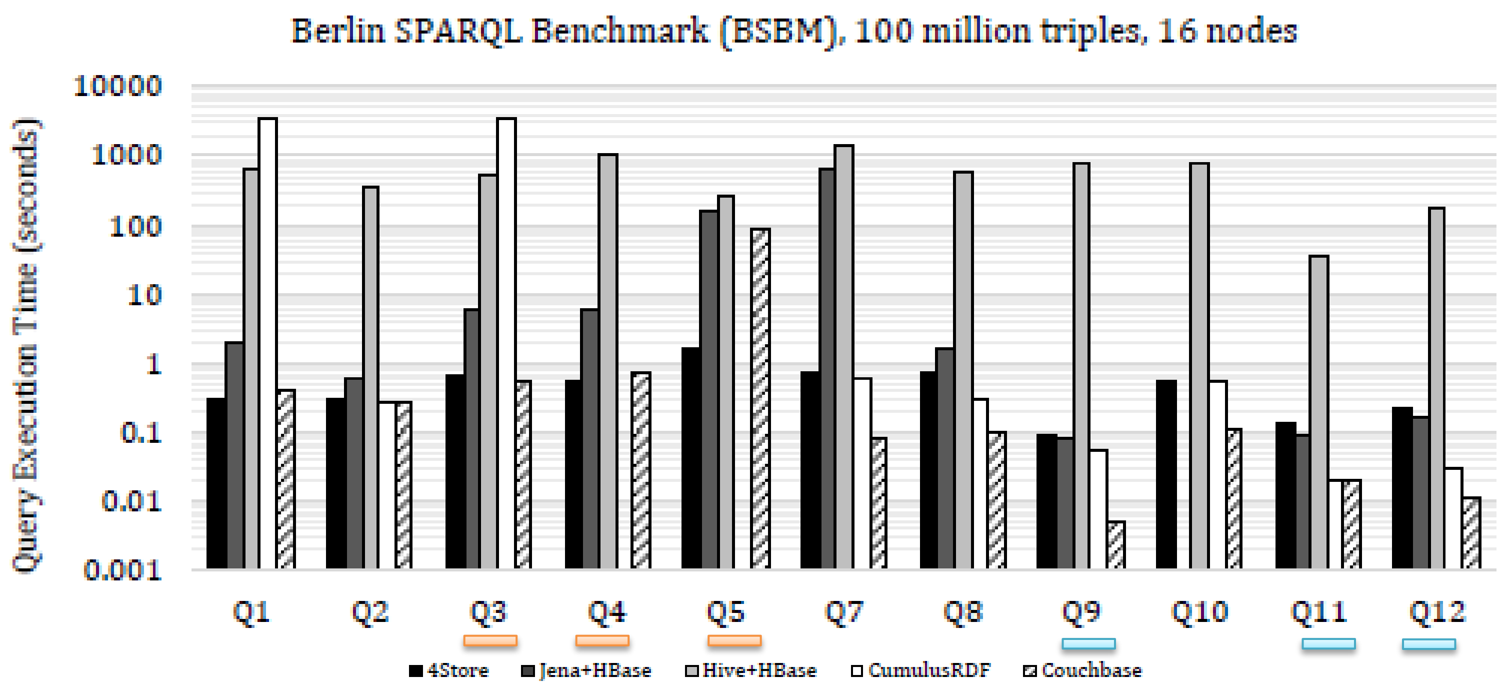


Figure taken from [4]

Outline

- Objectives
- Evaluated Systems
- Experiments and Results
- Conclusion
- Q&A

Conclusion



- Query time in NoSQL systems are competitive against native RDF stores
  - Simple workloads → good performance
  - Complex workloads→ poor performance

Figure taken from [5]

Conclusion

- Classical relational database query optimizations work well for RDF NoSQL systems
- Using MapReduce operations imposes latency

Thank you

- Q&A
- Discussion
  - The experiments are focused on read operation while write and update are also important in real situations. Although a system like Cassandra has high write throughput, we use several index tables that needs to be updated for writes. Will studying write and update may affect the conclusion?
  - Most of mentioned systems, store RDF data in multiple tables. Will this be a problem for data consistency?

References

[1] S. Harris, N. Lamb, and N. Shadbolt, “4store: The design and implementation of a clustered rdf store,” presented at the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009), 2009.

[2] P. Yuan, C. Xie, H. Jin, L. Liu, G. Yang, and X. Shi, “Dynamic and fast processing of queries on large-scale RDF data,” *Knowl Inf Syst*, vol. 41, no. 2, pp. 311–334, Jan. 2014.

[3] http://rdf4j.org/

[4] The figures in result section are reproduced from the published data set, retrieved from web.archive.com

[5] P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. L. Keppmann, D. Miranker, J. F. Sequeda, and M. Wylot, “NoSQL Databases for RDF: An Empirical Evaluation,” in *The Semantic Web – ISWC 2013*, H. Alani, L. Kagal, A. Fokoue, P. Groth, C. Biemann, J. X. Parreira, L. Aroyo, N. Noy, C. Welty, and K. Janowicz, Eds. Springer Berlin Heidelberg, 2013, pp. 310–325.