IBM Research

The case for storing RDF in a relational database

Kavitha Srinivas, IBM Research

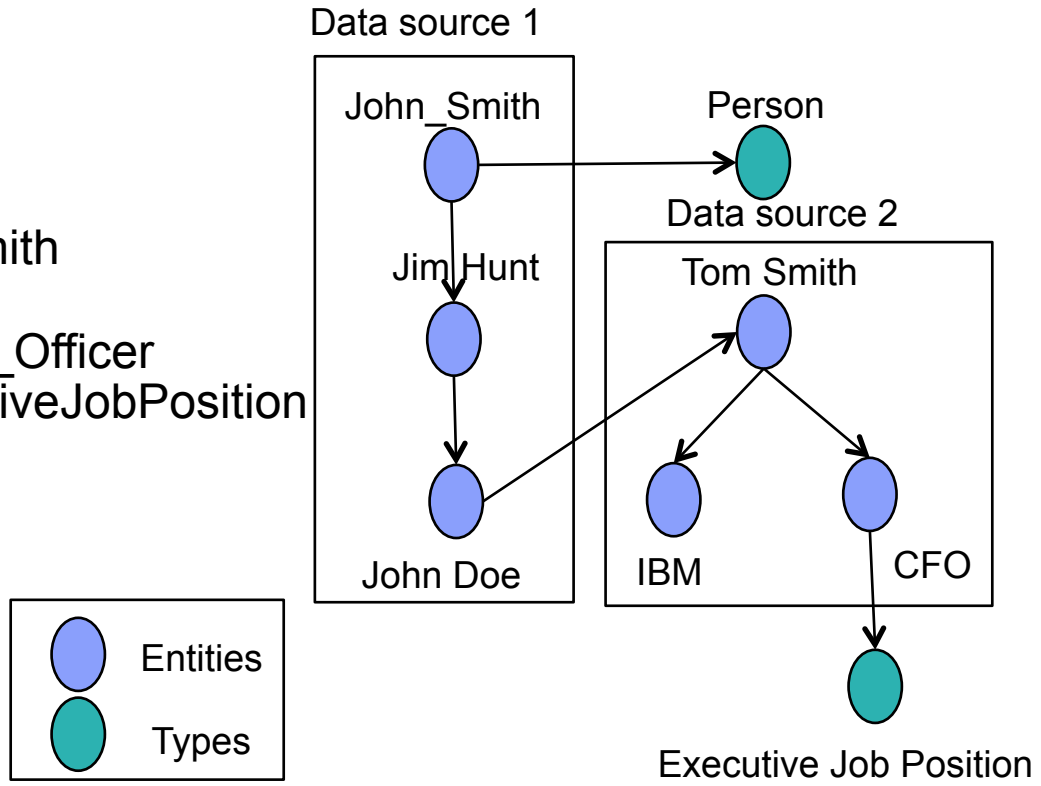
e business on demand.

What is RDF? In short a labeled directed graph

- Binary relationships between entities
- Entities mapped to semantic types
- Relationships between entities are labeled

John_Smith type Person
John_Smith hasReport Jim_Hunt
Jim_Hunt hasReport John_Doe
John_Doe hasContactWith Tom_Smith
Tom_Smith worksFor IBM
Tom_Smith jobTitle Chief_Financial_Officer
Chief_Financial_Officer type ExecutiveJobPosition

Different ‘types’ form a variable ‘schema’



RDF versus graph stores

Graph stores are a broader category for persisting graphs. Not all graph stores support RDF, which is a W3C standard.

Graph stores (such as neo4j) have custom APIs that provide a procedural mechanism to traverse graphs. The traversal is stepwise.

RDF stores have a declarative W3C standard query language called SPARQL to access subgraphs of interest.

RDF versus other 'schema-less' stores

'Schema-less' (such as Hbase, BigTable, Cassandra) stores tend to store sets of values associated with a key. E.g.,

John_Smith type Person
John_Smith hasReport Jim_Hunt
JimHunt hasReport John_Doe
John_Doe hasContactWith Tom_Smith
…

can be represented as:

Variable columns

Key	Type	hasReport	hasContact
John_Smith	Person	Jim_Hunt	
Jim_Hunt	hasReport	John_Doe	
John_Doe			Tom_Smith

Can store properties for a node in a graph. But no JOIN functionality. E.g., can't ask who in John Smith's reports has contactWith someone who works at IBM. No ability to traverse paths in a graph.

What do SPARQL queries look like?

Basically, mechanisms to find subgraph patterns

Find me a path from John Smith to his reports who have a contact with an executive at IBM

John_Smith hasReport ?z

?z hasContactWith ?x

?x worksFor IBM

?x jobTitle ?y

?y type ExecutiveJobPosition

?x, ?y ?z are variables that get bound to entities. Traversal in this form requires multiple joins.

So who needs RDF anyway?

Three major use cases for RDF, mainly because RDF allows complex queries across data with variable schema.

1. Data integration/Tool integration. Each tool has its own data model, each model's schema evolves differently with different types and properties.
2. Unstructured data access. Metadata generated by extractors for videos/text/images has different types and relations (based on the extractor).
3. Collaboratively developed repositories of knowledge. E.g. Wikipedia/Dbpedia, Freebase have types and properties that evolve as users add entities into the system.

Why build RDF over relational?

Key requirements:

1. Transactional support. Eventual consistency is sufficient in most cases.
2. Concurrent access. This is where the open source systems that our internal projects had used were weak.
3. Security and Access control. (a) Graph level access control (b) specialized predicates determining access.

Ride on top of relational systems' existing enterprise capabilities instead of reinventing the wheel.

- ACID, Security, Backup/recovery, compression, load balancing & parallel execution.

Challenges in laying out RDF in relational

Properties of RDF data:

1. Schema Scale. Number of entity types (tables) and properties (columns) are huge, and usually don't fit in relational constraints.
2. Schema Variability. Number of entity types and properties per column are constantly changing.
3. Data Sparsity. Each type can have thousands of properties, but not every entity of that type has all properties.

Standard Approach to RDF in Relational

Source	Edge Label	Target
John_Smith	Type	Person
John_Smith	hasReport	Jim_Hunt
John_Smith	hasTitle	CEO

Handles sparsity, schema scale and variability

A key problem with this approach is many joins are needed even if multiple properties about a given node is being accessed (so called ‘star queries’). E.g.:

John_Smith type ?y

John_Smith hasReport ?z

John_Smith hasTitle ?x

Requires joins to answer the query. In a relational system, this would be a single row lookup. Subqueries of most complex SPARQL queries are “stars”.

Stars in a complex SPARQL query

Find me a path from John Smith to his reports who have a contact with an executive at IBM

John_Smith hasReport ?z

?z hasContactWith ?x

?x worksFor IBM

?x jobTitle ?y

?y type ExecutiveJobPosition

Even in this complex query, queries about ?x's multiple properties are a 'star pattern'.

DB2RDF: Laying out RDF in relational

Key idea: Optimize star queries by using the row to store as many property-value pairs of a given node as possible.

To handle data sparsity, and evolving variable schema, each property is assigned to one or more columns based on:

1. Hash functions if no data characteristics are known. Multiple hashes reduce collisions, or we spill to a new row.
2. Property correlations if a sample of the data is known.

Source	Col 1	Col 2	Col 3	Col 4
John_Smith	Type	Person	hasReport	Jim_Hunt
Jim_Hunt			hasReport	John_Doe
John_Doe			hasContact	Tom_Smith

Primary Hashtable

hasReport & hasContact never co-occur together so they can be assigned to the same column

DB2RDF: Handling multi valued properties

Key idea: Use a key in primary hashtable to point to multi values in a different table

Source	Col 1	Col 2	Col 3	Col 4
John_Smith	Type	Person	hasReport	listId:1
Jim_Hunt			hasReport	John_Doe
John_Doe			hasContact	Tom_Smith

Primary Hashtable

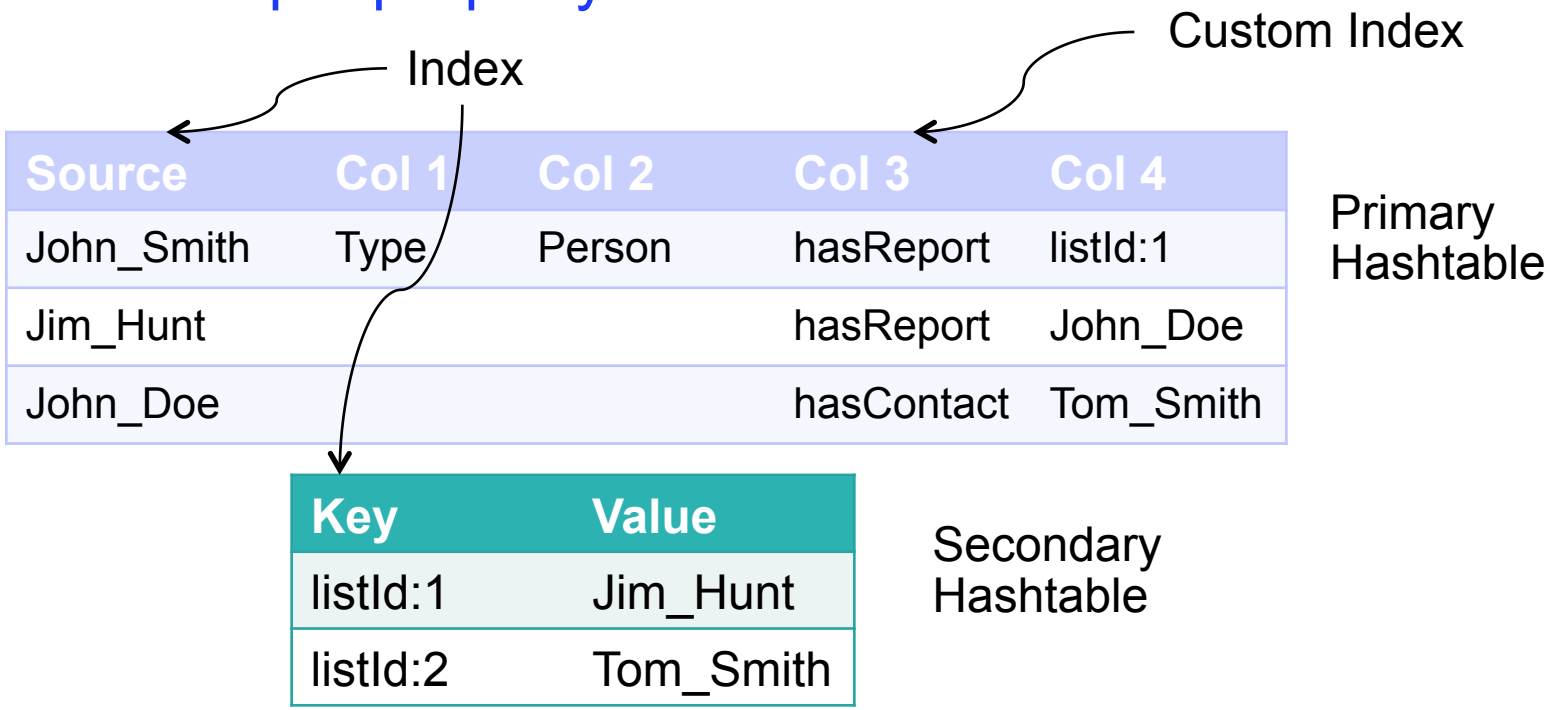
Key	Value
listId:1	Jim_Hunt
listId:2	Tom_Smith

Secondary Hashtable

Multi valued property access needs joins. Storing the presence of the multi valued property is important. It often gives us selectivity in star queries.

Indexing

- The source and key columns are indexed automatically, like most key→multiple value systems.
- Predicate indexes can be selectively added based on the query workload. Customized indexes for specific predicates is a unique property of DB2RDF.



Security and Access Control – Method 1

Access control for RDF exploits DB2's fine grained access control (FGAC) facility. Granularity of control is for a set of triples that are in the same graph

Source	Graph	PID	PCPID	Col 1	Col 2	Col 3	Col 4
John_Smith	g1	1	2	type	Patient	hasSSN	0123-456-89
Jim_Hunt	g2	2	2	type	Patient	hasSSN	0245-361-99
John_Doe	g3	3	3	type	Patient		

Goal: Let patients see their own data, let physicians see their patients' data.

- Segregate information for each patient into different graphs.
- Provide system predicates to the DB2RDF store so each predicate gets a dedicated column which can be used for FGAC.
- Use DB2 to configure rules to specify access to the row by role and identity of 'SESSION USER'.

Security and Access Control – Method 2

Handle security at an application level, pass in values for system predicates (PID and PCPID) in QueryContext that are valid for access.

Source	Graph	PID	PCPID	Col 1	Col 2	Col 3	Col 4
John_Smith	g1	1	2	type	Patient	hasSSN	0123-456-89
Jim_Hunt	g2	2	2	type	Patient	hasSSN	0245-361-99
John_Doe	g3	3	3	type	Patient		

Limitations:

- Less secure
- Onus for applying rules based on login and roles is in the application.
- Only supports equality or IN clauses for the system predicates.

Features in DB2RDF

- Data Access Operations
  - Jena API support (Jena ARQ for query, Jena for insert/update)
  - HTTP based SPARQL query through Joseki
- Query support: SPARQL 1.0 with subsets of SPARQL 1.1 (aggregation, subqueries)
- Reorganization facility: Better layout of data to improve performance.
- Availability: DB2Express (free version)

For an overview:

https://www.ibm.com/developerworks/mydeveloperworks/blogs/nlp/resource/DB2_NoSQLGraphStore.pdf?lang=en