An Overview of Graph Data Management and Analysis

M. Tamer Özsu

University of Waterloo
David R. Cheriton School of Computer Science

University of Waterloo
Graph Data are Very Common
Graph Data are Very Common

Social networks
Graph Data are Very Common

Trade volumes and connections

World Trade 1994
Residuals Model 1

Graph representation of trade volumes and connections.
Graph Data are Very Common

Graph Data are Very Common
Outline

1. Introduction – Graph Types

2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics

3. RDF Graph Querying
   - Data Warehousing
   - Distributed SPARQL Execution
   - Linked Object Data Querying
Outline

1. Introduction – Graph Types

2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics

3. RDF Graph Querying
   - Data Warehousing
   - Distributed SPARQL Execution
   - Linked Object Data Querying
Graph Types

Property graph
Graph Types

Property graph

- Workload: Online queries and analytic workloads
- Query execution: Varies

RDF graph

- Workload: SPARQL queries
- Query execution: subgraph matching by homomorphism
Outline

1. Introduction – Graph Types

2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics

3. RDF Graph Querying
   - Data Warehousing
   - Distributed SPARQL Execution
   - Linked Object Data Querying
Outline

1. Introduction – Graph Types

2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics

3. RDF Graph Querying
   - Data Warehousing
   - Distributed SPARQL Execution
   - Linked Object Data Querying
Classification

[Ammar and Özsu, 2015]

- Graph Dynamism
- Algorithm Types
- Workload Types
Focus here is on the dynamism of the graphs in whether or not they change and how they change.
Focus here is on the dynamism of the graphs in whether or not they change and how they change.

Focus here is on the how algorithms behave as their input changes.
Focus here is on the dynamism of the graphs in whether or not they change and how they change.

Focus here is on the how algorithms behave as their input changes.

The types of workloads that the approaches are designed to handle.
Graphs do not change or we are not interested in their changes – only a snapshot is considered.
Graphs do not change or we are not interested in their changes – only a *snapshot* is considered.

Graphs change and we are interested in their changes.
Graph Dynamism

- Static Graphs
- Dynamic Graphs
- Streaming Graphs
- Evolving Graphs

Algorithm Types

- Offline
- Online
- Streaming
- Incremental
- Dynamic
- Batch

Workload Types

- Online Queries
- Analytics Workloads

Graphs do not change or we are not interested in their changes – only a *snapshot* is considered.

Graphs change and we are interested in their changes.

Dynamic graphs with high velocity changes – not possible to see the entire graph at once.
Classification

- **Graph Dynamism**
  - Static Graphs
  - Dynamic Graphs
  - Streaming Graphs
  - Evolving Graphs

- **Algorithm Types**
  - Offline
  - Online
  - Incremental
  - Dynamic
  - Batch

- **Workload Types**
  - Online Queries
  - Analytics Workloads

---

Graphs do not change or we are not interested in their changes – only a *snapshot* is considered.

Graphs change and we are interested in their changes.

Dynamic graphs with high velocity changes – not possible to see the entire graph at once.

Dynamic graphs with unknown changes – requires re-discovery of the graph (e.g., LOD).
Computation accesses a portion of the graph and the results are computed for a subset of vertices; e.g., point-to-point shortest path, subgraph matching, reachability, SPARQL.
Computation accesses a portion of the graph and the results are computed for a subset of vertices; e.g., point-to-point shortest path, subgraph matching, reachability, SPARQL.

Computation accesses the entire graph and may require multiple iterations; e.g., PageRank, clustering, graph colouring, all pairs shortest path.
Classification

[Ammar and Özsu, 2015]

Graph Dynamism
- Static Graphs
- Dynamic Graphs
- Streaming Graphs
- Evolving Graphs

Algorithm Types
- Offline
- Online
  - Streaming
  - Incremental
- Dynamic
  - Batch
  - Dynamic

Workload Types
- Online Queries
- Analytics Workloads
Classification

Graph Dynamism
- Static Graphs
- Dynamic Graphs
- Streaming Graphs
- Evolving Graphs

Algorithm Types
- Offline
- Online
- Dynamic

Workload Types
- Online Queries
- Analytics Workloads

Sees the entire input in advance.
**Classification**

- **Graph Dynamism**: Static Graphs, Dynamic Graphs, Streaming Graphs, Evolving Graphs
- **Algorithm Types**: Offline, Online, Streaming, Incremental
- **Workload Types**: Online Queries, Analytics Workloads

Sees the entire input in advance.

Sees the input piece-meal as it executes.
Classification

[Ammar and Özsu, 2015]

Graph Dynamism

- Static Graphs
- Dynamic Graphs
- Streaming Graphs
- Evolving Graphs

Algorithm Types

- Offline
- Online
- Streaming
- Incremental

Workload Types

- Online Queries
- Analytics Workloads

Sees the entire input in advance.

Sees the input piece-meal as it executes.

One-pass online algorithm with limited memory.
Classification

[Ammar and Özsu, 2015]

Graph Dynamism
- Static Graphs
- Dynamic Graphs
- Streaming Graphs
- Evolving Graphs

Algorithm Types
- Offline
- Online
- Streaming
- Incremental

Workload Types
- Online Queries
- Analytics Workloads

- Sees the entire input in advance.
- Sees the input piece-meal as it executes.
- One-pass online algorithm with limited memory.
- Online algorithm with some info about forthcoming input.
Classification

Graph Dynamism
- Static Graphs
- Dynamic Graphs
  - Streaming Graphs
  - Evolving Graphs

Algorithm Types
- Offline
- Online
- Streaming
  - Incremental
  - Batch
- Dynamic

Workload Types
- Online Queries
- Analytics
  - Sees the entire input in advance, which may change; answers computed as change occurs.

Sees the entire input in advance.
Sees the input piece-meal as it executes.
One-pass online algorithm with limited memory.
Online algorithm with some info about forthcoming input.
Classification

Graph Dynamism
- Static Graphs
- Dynamic Graphs
- Streaming Graphs
- Evolving Graphs

Algorithm Types
- Offline
- Online
- Streaming
- Incremental
- Batch
- Dynamic

Workload Types
- Online Queries
- Analytics

Sees the entire input in advance.
Sees the input piece-meal as it executes.
One-pass online algorithm with limited memory.
Online algorithm with some info about forthcoming input.
Similar to dynamic, but computation happens in batches of
Example Design Points

Graph Dynamism
- Static Graphs
- Dynamic Graphs
- Streaming Graphs
- Evolving Graphs

Algorithm Types
- Offline
- Online
- Dynamic
- Streaming
- Incremental
- Batch
- Dynamic

Workload Types
- Online Queries
- Analytics Workloads

Compute the query result/perform analytic computation over the graph as it exists.
Example Design Points

Graph Dynamism
- Static Graphs
- Dynamic Graphs
- Streaming Graphs
- Evolving Graphs

Algorithm Types
- Offline
- Online
- Dynamic
- Streaming
- Incremental

Workload Types
- Online Queries
- Analytics Workloads

Compute the query result/perform analytic computation over the graph as it is revealed.
Example Design Points

- Graph Dynamism
  - Static Graphs
  - Dynamic Graphs
  - Streaming Graphs
  - Evolving Graphs

- Algorithm Types
  - Offline
  - Online
    - Streaming Incremental
    - Batch Dynamic

- Workload Types
  - Online Queries
  - Analytics Workloads

Compute the query result/perform analytic computation on each snapshot from scratch.
Example Design Points

Graph Dynamism
- Static Graphs
- Dynamic Graphs
- Streaming Graphs
- Evolving Graphs

Algorithm Types
- Offline
- Online
  - Streaming
  - Incremental
  - Batch

Workload Types
- Dynamic
  - Online Queries
  - Analytics Workloads

Continuously compute the query result/perform analytic computation as the input changes.
Compute the query result/perform analytic computation after a batch of input changes.
Dynamic (or batch-dynamic) algorithms do not make sense for static graphs.
## Graph Processing Systems

<table>
<thead>
<tr>
<th>System</th>
<th>Memory/Disk</th>
<th>Architecture</th>
<th>Computing paradigm</th>
<th>Declarative Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>Disk</td>
<td>Parallel/Distributed</td>
<td>MapReduce</td>
<td>×</td>
</tr>
<tr>
<td>Halloop</td>
<td>Disk</td>
<td>Parallel/Distributed</td>
<td>MapReduce</td>
<td>×</td>
</tr>
<tr>
<td>Pegasus</td>
<td>Disk</td>
<td>Parallel/Distributed</td>
<td>MapReduce</td>
<td>×</td>
</tr>
<tr>
<td>Pregel/Giraph</td>
<td>Memory</td>
<td>Parallel/Distributed</td>
<td>Vertex-Centric</td>
<td>×</td>
</tr>
<tr>
<td>GraphLab</td>
<td>Memory</td>
<td>Parallel/Distributed</td>
<td>Vertex-Centric</td>
<td>×</td>
</tr>
<tr>
<td>GraphChi</td>
<td>Disk</td>
<td>Single machine</td>
<td>Vertex-Centric</td>
<td>×</td>
</tr>
<tr>
<td>GraphX</td>
<td>Disk</td>
<td>Single machine</td>
<td>Edge-Centric</td>
<td>×</td>
</tr>
<tr>
<td>TurboGraph</td>
<td>Disk</td>
<td>Single machine</td>
<td>Vertex-Centric</td>
<td>×</td>
</tr>
<tr>
<td>Trinity</td>
<td>Memory</td>
<td>Parallel/Distributed</td>
<td>MapReduce/Vertex-Centric</td>
<td>✓ (TSL)</td>
</tr>
<tr>
<td>Titan</td>
<td>Disk</td>
<td>Parallel/Distributed</td>
<td>?</td>
<td>✓ (Gremlin)</td>
</tr>
<tr>
<td>Neo4J</td>
<td>Disk</td>
<td>Single machine</td>
<td>Procedural/Linked-list</td>
<td>✓ (Cypher)</td>
</tr>
</tbody>
</table>
Graph Workloads

Online graph querying
- Reachability
- Single source shortest-path
- Subgraph matching
- SPARQL queries

Offline graph analytics
- PageRank
- Clustering
- Strongly connected components
- Diameter finding
- Graph colouring
- All pairs shortest path
- Graph pattern mining
- Machine learning algorithms
  (Belief propagation, Gaussian non-negative matrix factorization)
Outline

1. Introduction – Graph Types

2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics

3. RDF Graph Querying
   - Data Warehousing
   - Distributed SPARQL Execution
   - Linked Object Data Querying
Reachability Queries

- **film_2014**
  - initial_release_date: "1980-05-23"
  - label: "The Shining"
  - (actor)
  - (director)
  - (relatedBook)
  - (based_near)

- **film_3418**
  - label: "The Passenger"
  - (actor)
  - (director)
  - (relatedBook)

- **film_1267**
  - label: "The Last Tycoon"
  - (actor)
  - (director)
  - (relatedBook)

- **books_0743424425**
  - rating: 4.7
  - (hasOffer)

- **geo_2635167**
  - name: "United Kingdom"
  - population: 6234847

- **film_2685**
  - label: "A Clockwork Orange"
  - (director)

- **film_424**
  - label: "Spartacus"
  - (director)

- **director_8476**
  - director_name: "Stanley Kubrick"

- **actor_29704**
  - actor_name: "Jack Nicholson"

- **actor_30013**

- **offers_0743424425amazonOffer**
  - (hasOffer)
Can you reach film_1267 from film_2014?
Is there a book whose rating is $> 4.0$ associated with a film that was directed by Stanley Kubrick?
Think of Facebook graph and finding friends of friends.
Outline

1 Introduction – Graph Types

2 Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics

3 RDF Graph Querying
   - Data Warehousing
   - Distributed SPARQL Execution
   - Linked Object Data Querying
A web page is important if it is pointed to by other important pages.

\[ r(P_i) = \sum_{P_j \in B_{P_i}} \frac{r(P_j)}{|F_{P_j}|} \]

\[ r(P_2) = \frac{r(P_1)}{2} + \frac{r(P_3)}{3} \]

\[ r_{k+1}(P_i) = \sum_{P_j \in B_{P_i}} \frac{r_k(P_j)}{|F_{P_j}|} \]

- \( B_{P_i} \): in-neighbours of \( P_i \)
- \( F_{P_i} \): out-neighbours of \( P_i \)
PageRank Computation

A web page is important if it is pointed to by other important pages.

\[ r_{k+1}(P_i) = \sum_{P_j \in B_{P_i}} \frac{r_k(P_j)}{|F_{P_j}|} \]

<table>
<thead>
<tr>
<th>Iteration 0</th>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Rank at Iter. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_0(P_1) = 1/6 )</td>
<td>( r_1(P_1) = 1/18 )</td>
<td>( r_2(P_1) = 1/36 )</td>
<td>5</td>
</tr>
<tr>
<td>( r_0(P_2) = 1/6 )</td>
<td>( r_1(P_2) = 5/36 )</td>
<td>( r_2(P_2) = 1/18 )</td>
<td>4</td>
</tr>
<tr>
<td>( r_0(P_3) = 1/6 )</td>
<td>( r_1(P_3) = 1/12 )</td>
<td>( r_2(P_3) = 1/36 )</td>
<td>5</td>
</tr>
<tr>
<td>( r_0(P_4) = 1/6 )</td>
<td>( r_1(P_4) = 1/4 )</td>
<td>( r_2(P_4) = 17/72 )</td>
<td>1</td>
</tr>
<tr>
<td>( r_0(P_5) = 1/6 )</td>
<td>( r_1(P_5) = 5/36 )</td>
<td>( r_2(P_5) = 11/72 )</td>
<td>3</td>
</tr>
<tr>
<td>( r_0(P_6) = 1/6 )</td>
<td>( r_1(P_6) = 1/6 )</td>
<td>( r_2(P_6) = 14/72 )</td>
<td>2</td>
</tr>
</tbody>
</table>

Iterative processing.
Some Alternative Computational Models for Offline Analytics

- **MapReduce**
  - map and reduce functions
  - Not suitable for iterative processing due to data movement at each stage
  - Need to save in storage system intermediate results of each iteration
Some Alternative Computational Models for Offline Analytics

- **MapReduce**
  - map and reduce functions
  - Not suitable for iterative processing due to data movement at each stage
  - Need to save in storage system intermediate results of each iteration

- **Vertex-centric paradigm**
  - Specify (a) the computation to be performed at each vertex, and (b) its communication with neighbour vertices
  - Designed specifically for interactive graph processing
  - Synchronous (e.g., Pregel, Giraph)

- Asynchronous (e.g., GraphLab)
Pregel-like systems are **BSP**, **vertex-centric** programs.
Pregel-like systems are **BSP**, **vertex-centric** programs.

Computation
Pregel-like systems are **BSP**, **vertex-centric** programs.
Pregel-like systems are **BSP**, **vertex-centric** programs.
Pregel-like systems are **BSP**, **vertex-centric** programs.
Pregel-like systems are **BSP**, *vertex-centric* programs.
Pregel-like systems are **BSP**, **vertex-centric** programs.

- “Think like a vertex”:
GraphLab features *asynchronous* execution:
- No communication barriers. ✓
- Uses the *most recent* vertex values. ✓
GraphLab (Asynchronous)

Implemented via distributed locking:
GraphLab (Asynchronous)

Implemented via distributed locking:

\[ \begin{array}{c}
V_0 \\
V_1 \quad V_2 \\
V_3 \quad V_4
\end{array} \]

© M. Tamer Özsu  
ADC PhD School  
(2015-06-04) 21 / 76
GraphLab (Asynchronous)

Implemented via distributed locking:
GraphLab (Asynchronous)

Implemented via distributed locking:

$\forall_0 \forall_1 \forall_2 \forall_3 \forall_4$
GraphLab (Asynchronous)

Implemented via distributed locking:

\[ v_0 \]
\[ v_1 \rightarrow v_0 \rightarrow v_2 \]
\[ v_3 \rightarrow v_0 \rightarrow v_4 \]
A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs; GraphLab scales better across more machines.
Summary of an Experiment

A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs;
   GraphLab scales better across more machines.

<table>
<thead>
<tr>
<th></th>
<th>64 machines</th>
<th>TW</th>
<th>UK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Giraph (byte array)</td>
<td>5.8GB</td>
<td>7.0GB</td>
<td></td>
</tr>
<tr>
<td>GraphLab (sync)</td>
<td>4.5GB</td>
<td>14GB</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>TW</th>
<th>16 machines</th>
<th>128 machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Giraph (byte array)</td>
<td>8.5GB</td>
<td>5.8GB</td>
<td></td>
</tr>
<tr>
<td>GraphLab (sync)</td>
<td>11GB</td>
<td>3.3GB</td>
<td></td>
</tr>
</tbody>
</table>
Summary of an Experiment

A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs; GraphLab scales better across more machines.

2. Distributed locking for asynchronous execution is not scalable – Performance degrades as more machines are used due to lock contention, termination scheme, lack of message batching.
A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs; GraphLab scales better across more machines.

2. Distributed locking for asynchronous execution is not scalable – Performance degrades as more machines are used due to lock contention, termination scheme, lack of message batching.

3. Graph storage should be memory and mutation efficient.
Summary of an Experiment

A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs; GraphLab scales better across more machines.

2. Distributed locking for asynchronous execution is not scalable – Performance degrades as more machines are used due to lock contention, termination scheme, lack of message batching.

3. Graph storage should be memory and mutation efficient.

<table>
<thead>
<tr>
<th>No Mutations</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte array</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Hash map</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>With Mutations (DMST)</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte array</td>
<td>☒</td>
<td>✔</td>
</tr>
<tr>
<td>Hash map</td>
<td>✔</td>
<td>✗</td>
</tr>
</tbody>
</table>
A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs; GraphLab scales better across more machines.
2. Distributed locking for asynchronous execution is not scalable – Performance degrades as more machines are used due to lock contention, termination scheme, lack of message batching.
3. Graph storage should be memory and mutation efficient.
4. Message *processing* optimizations are very important.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CPU</th>
<th>Memory</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>SSSP</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>WCC</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>DMST</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
</tr>
</tbody>
</table>
Outline

1. Introduction – Graph Types

2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics

3. RDF Graph Querying
   - Data Warehousing
   - Distributed SPARQL Execution
   - Linked Object Data Querying
RDF Introduction

- Everything is an uniquely named resource

http://data.linkedmdb.org/resource/actor/JN29704

© M. Tamer Özsu  ADC PhD School  (2015-06-04)  25 / 76
RDF Introduction

- Everything is an **uniquely** named resource
- Prefixes can be used to shorten the names

```
xmIn:yi=http://data.linkedmdb.org/resource/actor/
y:JN29704
y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"
y:TS2014:title "The Shining"
y:TS2014:releaseDate "1980-05-23"
y:TS2014:JN29704:movieActor
```
- Everything is an **uniquely** named resource
- Prefixes can be used to shorten the names
- Properties of resources can be defined

```
xmLns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704

y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"
```
RDF Introduction

- Everything is an uniquely named resource
- Prefixes can be used to shorten the names
- Properties of resources can be defined
- Relationships with other resources can be defined

```
xmIn:y=http://data.linkedmdb.org/resource/actor/
y:JN29704

y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"

JN29704:movieActor

y:TS2014

y:TS2014:title "The Shining"
y:TS2014:releaseDate "1980-05-23"
```

© M. Tamer Özsu
ADC PhD School (2015-06-04) 25 / 76
RDF Introduction

- Everything is an **uniquely** named resource
- Prefixes can be used to shorten the names
- Properties of resources can be defined
- Relationships with other resources can be defined
- Resource descriptions can be contributed by different people/groups and can be located anywhere in the web
  - Integrated web “database”

```xml
xmlns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704

y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"
y:JN29704:movieActor

y:TS2014:title "The Shining"
y:TS2014:releaseDate "1980-05-23"
```
RDF Data Model

- **Triple**: Subject, Predicate (Property), Object \((s, p, o)\)
  - **Subject**: the entity that is described (URI or blank node)
  - **Predicate**: a feature of the entity (URI)
  - **Object**: value of the feature (URI, blank node or literal)
- \((s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)\)
- Set of RDF triples is called an **RDF graph**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://...imdb.../29704">http://...imdb.../29704</a></td>
<td>movie:actor_name</td>
<td>“Jack Nicholson”</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### RDF Example Instance

Prefixes:  
- `mdb=http://data.linkedmdb.org/resource/`  
- `geo=http://sws.geonames.org/`  
- `bm=http://wifo5-03.informatik.uni-mannheim.de/bookmashup/`  
- `lexvo=http://lexvo.org/id/`  
- `wp=http://en.wikipedia.org/wiki/`

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mdb: film/2014</code></td>
<td>rdfs:label</td>
<td>&quot;The Shining&quot;</td>
</tr>
<tr>
<td><code>mdb:film/2014</code></td>
<td>movie:director</td>
<td><code>mdb:director/8476</code></td>
</tr>
<tr>
<td><code>mdb:film/2014</code></td>
<td>movie:music_contributor</td>
<td><code>mdb: music_contributor/4110</code></td>
</tr>
<tr>
<td><code>mdb:film/2014</code></td>
<td>foaf:based_near</td>
<td><code>geo:2635167</code></td>
</tr>
<tr>
<td><code>mdb:film/2685</code></td>
<td>movie:relatedBook</td>
<td><code>bm:074324425</code></td>
</tr>
<tr>
<td><code>mdb:director/8476</code></td>
<td>movie:director_name</td>
<td>&quot;Stanley Kubrick&quot;</td>
</tr>
<tr>
<td><code>mdb:film/2685</code></td>
<td>movie:director</td>
<td><code>mdb:director/8476</code></td>
</tr>
<tr>
<td><code>mdb:film/2685</code></td>
<td>rdfs:label</td>
<td>&quot;A Clockwork Orange&quot;</td>
</tr>
<tr>
<td><code>mdb:film/424</code></td>
<td>movie:director</td>
<td><code>mdb:director/8476</code></td>
</tr>
<tr>
<td><code>mdb:actor/29704</code></td>
<td>movie:actor_name</td>
<td>&quot;Jack Nicholson&quot;</td>
</tr>
<tr>
<td><code>mdb:film/1267</code></td>
<td>movie:actor</td>
<td><code>mdb:actor/29704</code></td>
</tr>
<tr>
<td><code>mdb:film/1267</code></td>
<td>rdfs:label</td>
<td>&quot;The Last Tycoon&quot;</td>
</tr>
<tr>
<td><code>mdb:film/3418</code></td>
<td>movie:actor</td>
<td><code>mdb:actor/29704</code></td>
</tr>
<tr>
<td><code>mdb:film/3418</code></td>
<td>rdfs:label</td>
<td>&quot;The Passenger&quot;</td>
</tr>
<tr>
<td><code>geo:2635167</code></td>
<td>gn:name</td>
<td>&quot;United Kingdom&quot;</td>
</tr>
<tr>
<td><code>geo:2635167</code></td>
<td>gn:population</td>
<td>62348447</td>
</tr>
<tr>
<td><code>geo:2635167</code></td>
<td>gn:wikipediaArticle</td>
<td><code>wp:United_Kingdom</code></td>
</tr>
<tr>
<td><code>bm:books/0743424425</code></td>
<td>dc:creator</td>
<td><code>bm:persons/Stephen+King</code></td>
</tr>
<tr>
<td><code>bm:books/0743424425</code></td>
<td>rev:rating</td>
<td>4.7</td>
</tr>
<tr>
<td><code>bm:books/0743424425</code></td>
<td>scom:hasOffer</td>
<td><code>bm:offers/0743424425amazonOffer</code></td>
</tr>
<tr>
<td><code>lexvo:iso639-3/eng</code></td>
<td>rdfs:label</td>
<td>&quot;English&quot;</td>
</tr>
<tr>
<td><code>lexvo:iso639-3/eng</code></td>
<td>lvont:usedLn</td>
<td><code>lexvo:iso3166/CA</code></td>
</tr>
<tr>
<td><code>lexvo:iso639-3/eng</code></td>
<td>lvont:usesScript</td>
<td><code>lexvo:script/Latn</code></td>
</tr>
</tbody>
</table>
RDF Query Model – SPARQL Protocol and RDF Query Language

Given $U$ (set of URIs), $L$ (set of literals), and $V$ (set of variables), a SPARQL expression is defined recursively:
- an atomic triple pattern, which is an element of
  
  \[(U \cup V) \times (U \cup V) \times (U \cup V \cup L)\]
- $?x rdfs:label “The Shining”
- $P$ \texttt{FILTER} $R$, where $P$ is a graph pattern expression and $R$ is a built-in SPARQL condition (i.e., analogous to a SQL predicate)
  - $?x rev:rating ?p \texttt{FILTER}(?p > 3.0)
- $P1$ \texttt{AND/OPT/UNION} $P2$, where $P1$ and $P2$ are graph pattern expressions

Example:

\begin{verbatim}
SELECT ?name
WHERE {
  ?d movie:director_name "Stanley Kubrick".
  \texttt{FILTER} (?r > 4.0)
}
\end{verbatim}
SELECT ?name
WHERE {
  ?d movie:director_name "Stanley Kubrick".
  FILTER(?r > 4.0)
}

FILTER(?r > 4.0)
1. Introduction – Graph Types

2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics

3. RDF Graph Querying
   - Data Warehousing
   - Distributed SPARQL Execution
   - Linked Object Data Querying
SELECT ?name
WHERE {
  ?d movie:director_name "Stanley Kubrick" .
  FILTER (?r > 4.0)
}

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdb:film/2014</td>
<td>rdfs:label</td>
<td>&quot;The Shining&quot;</td>
</tr>
<tr>
<td>mdb:film/2014</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2014</td>
<td>foaf:based_near</td>
<td>geo:2635167</td>
</tr>
<tr>
<td>mdb:director/8476</td>
<td>movie:director_name</td>
<td>“Stanley Kubrick”</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>rdfs:label</td>
<td>“A Clockwork Orange”</td>
</tr>
<tr>
<td>mdb:film/424</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/424</td>
<td>rdfs:label</td>
<td>“Spartacus”</td>
</tr>
<tr>
<td>mdb:actor/29704</td>
<td>movie:actor</td>
<td>mdb:actor/29704</td>
</tr>
<tr>
<td>mdb:film/1267</td>
<td>movie:actor</td>
<td>&quot;The Last Tycoon&quot;</td>
</tr>
<tr>
<td>mdb:film/1267</td>
<td>rdfs:label</td>
<td>mdb:actor/29704</td>
</tr>
<tr>
<td>mdb:film/3418</td>
<td>movie:actor</td>
<td>&quot;The Passenger&quot;</td>
</tr>
<tr>
<td>mdb:film/3418</td>
<td>rdfs:label</td>
<td>&quot;United Kingdom&quot;</td>
</tr>
<tr>
<td>geo:2635167</td>
<td>gn:name</td>
<td>62348447</td>
</tr>
<tr>
<td>geo:2635167</td>
<td>gn:population</td>
<td></td>
</tr>
<tr>
<td>geo:2635167</td>
<td>gn:wikiArticle</td>
<td></td>
</tr>
<tr>
<td>bm:books/0743424425</td>
<td>dc:creator</td>
<td>bm:persons/Stephen+King</td>
</tr>
<tr>
<td>bm:books/0743424425</td>
<td>rev:rating</td>
<td>4.7</td>
</tr>
<tr>
<td>bm:books/0743424425</td>
<td>scom:hasOffer</td>
<td>bm:offers/0743424425amazonOffer</td>
</tr>
<tr>
<td>lexvo:iso639-3/eng</td>
<td>rdfs:label</td>
<td>lexvo:iso3166/CA</td>
</tr>
<tr>
<td>lexvo:iso639-3/eng</td>
<td>lvont:usesIn</td>
<td>lexvo:script/Latn</td>
</tr>
</tbody>
</table>

Easy to implement but too many self-joins!
SELECT ?name
WHERE {
  ?d movie:director_name "Stanley Kubrick".
  FILTER (?r > 4.0)
}

SELECT T1.object
FROM T as T1, T as T2, T as T3, T as T4, T as T5
WHERE T1.p="rdfs:label"
  AND T2.p="movie:relatedBook"
  AND T3.p="movie:director"
  AND T4.p="rev:rating"
  AND T5.p="movie:director_name"
  AND T1.s=T2.s
  AND T1.s=T3.s
  AND T2.o=T4.s
  AND T3.o=T5.s
  AND T4.o > 4.0
  AND T5.o="Stanley Kubrick"
Naïve Triple Store Design

SELECT ?name
WHERE {
  ?m rdfs:label ?name .
  ?m movie:director_name "Stanley Kubrick" .
  ?b rev:rating ?r .
  FILTER (?r > 4.0)
}

Easy to implement but too many self-joins!
### Property Tables

- **Grouping by entities; Jena** [Wilkinson, 2006], **DB2-RDF** [Bornea et al., 2013]

- **Clustered property table:** group together the properties that tend to occur in the same (or similar) subjects

- **Property-class table:** cluster the subjects with the same type of property into one property table

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdb:film/2014</td>
<td>movie:director</td>
<td>mb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>rdfs:label</td>
<td>“A Clockwork Orange” mb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>movie:director</td>
<td>mb:director/8476</td>
</tr>
<tr>
<td>mdb:actor/29704</td>
<td>rdfs:label</td>
<td>“Jack Nicholson”</td>
</tr>
<tr>
<td></td>
<td>movie:actor_name</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>refs:label</th>
<th>movie:director</th>
</tr>
</thead>
<tbody>
<tr>
<td>mob:film/2685</td>
<td>“The Clockwork Orange”</td>
<td>mb:director/8476</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>movie:actor_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mb:actor</td>
<td>“Jack Nicholson”</td>
</tr>
</tbody>
</table>
Property Tables

- Grouping by entities; Jena [Wilkinson, 2006], DB2-RDF [Bornea et al., 2013]
- Clustered property table: group together the properties that tend to occur in the same (or similar) subjects

### Advantages

- Fewer joins
- If the data is structured, we have a relational system – similar to normalized relations

<table>
<thead>
<tr>
<th>Subject</th>
<th>refs:label</th>
<th>movie:director</th>
</tr>
</thead>
<tbody>
<tr>
<td>mob:film/2685</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>movie:actor_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdb:actor/29704</td>
<td>“Jack Nicholson”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>movie:director/8476</th>
</tr>
</thead>
<tbody>
<tr>
<td>mob:director/8476</td>
<td>“A Clockwork Orange”</td>
</tr>
</tbody>
</table>

© M. Tamer Özsu
ADC PhD School
(2015-06-04) 33 / 76
Property Tables

- Grouping by entities; Jena [Wilkinson, 2006], DB2-RDF [Bornea et al., 2013]

- *Clustered property table*: group together the properties that tend to occur as a pair or a set on the same subject.

### Advantages

- Fewer joins
- If the data is structured, we have a relational system – similar to normalized relations

### Disadvantages

- Potentially a lot of NULLs
- Clustering is not trivial
- Multi-valued properties are complicated

---

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdb:film/2014</td>
<td>rdfs:label</td>
<td>&quot;The Shining&quot;</td>
</tr>
<tr>
<td>mdb:film/2014</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mob:film/2014</td>
<td>&quot;The Shining&quot;</td>
<td>mob:director/8476</td>
</tr>
<tr>
<td>mob:film/2685</td>
<td>&quot;The Clockwork Orange&quot;</td>
<td>mob:director/8476</td>
</tr>
</tbody>
</table>

© M. Tamer Özsu
ADC PhD School (2015-06-04) 33 / 76
Grouping by properties: For each property, build a two-column table, containing both subject and object, ordered by subjects [Abadi et al., 2007, 2009].

Also called vertical partitioned tables.

$n$ two column tables ($n$ is the number of unique properties in the data)

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdb:film/2014</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>movie:director</td>
<td>“A Clockwork Orange”</td>
</tr>
<tr>
<td>mdb:actor/29704</td>
<td>movie:actor_name</td>
<td>“Jack Nicholson”</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Binary Tables

- Grouping by properties: For each property, build a two-column table, containing both subject and object values related by subject. [Abadi et al., 2007, 2009]

Also called vertical partitioned tables

\[ n \]

\( n \) is the number of unique properties in the data

<table>
<thead>
<tr>
<th>Subject</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mob:film/2014</td>
<td>&quot;The Shining&quot;</td>
</tr>
<tr>
<td>mob:film/2685</td>
<td>&quot;The Clockwork Orange&quot;</td>
</tr>
</tbody>
</table>

Advantages

- Supports multi-valued properties
- No NULLs
- No clustering
- Read only needed attributes (i.e. less I/O)
- Good performance for subject-subject joins
Binary Tables

- Grouping by properties: For each property, build a two-column table, containing both subject and object values ordered by subject [Abadi et al., 2007, 2009]
- Also called vertical partitioned tables

```
<table>
<thead>
<tr>
<th>Subject</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdb:film/2014</td>
<td>rdfs:label</td>
</tr>
<tr>
<td></td>
<td>&quot;The Shining&quot;</td>
</tr>
<tr>
<td>mdb:film/2014</td>
<td>movie:director</td>
</tr>
<tr>
<td></td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>movie:director</td>
</tr>
<tr>
<td></td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>rdfs:label</td>
</tr>
<tr>
<td></td>
<td>&quot;A Clockwork Orange&quot;</td>
</tr>
<tr>
<td>mdb:actor/29704</td>
<td>movie:actor:name</td>
</tr>
<tr>
<td></td>
<td>&quot;Jack Nicholson&quot;</td>
</tr>
</tbody>
</table>
```

**Advantages**

- Supports multi-valued properties
- No NULLs
- No clustering
- Read only needed attributes (i.e. less I/O)
- Good performance for subject-subject joins

**Disadvantages**

- Not useful for subject-object joins
- Expensive inserts
Graph-based Approach

- Answering SPARQL query \( \equiv \) subgraph matching using homomorphism
- gStore [Zou et al., 2011, 2014], chameleon-db [Aluç et al., 2013]
Graph-based Approach

- Answering SPARQL query \(\equiv\) subgraph matching using homomorphism
- gStore [Zou et al., 2011, 2014], chameleon-db [Aluç et al., 2013]

Advantages

- Maintains the graph structure
- Full set of queries can be handled
Graph-based Approach

- Answering SPARQL query ≡ subgraph matching using homomorphism
- gStore [Zou et al., 2011, 2014], chameleon-db [Aluç et al., 2013]

Advantages

▶ Maintains the graph structure
▶ Full set of queries can be handled

Disadvantages

▶ Graph pattern matching is expensive

© M. Tamer Özsu
ADC PhD School (2015-06-04) 35 / 76
The online query evaluation process consists of two steps: computing eS_{ig(e)} and the encoding strategy is analogue over RDF graph.

Let s_{ig(e)} denote the neighbor vertex label (i.e., v_a ... v_n|v_m...v_n). Using an appropriate hash function, we set m out of M bits in s_{ig(e)} to be '1'.

Also present some system-oriented optimization, such as index query processing by subgraph matching over the signature according to our framework in Section II, we solve the SPARQL gStore system; full details are given in elsewhere [5], [6].
General Approach:

- Work directly on the RDF graph and the SPARQL query graph
- Use a signature-based encoding of each entity and class vertex to speed up matching
- Filter-and-evaluate
  - Use a false positive algorithm to prune nodes and obtain a set of candidates; then do more detailed evaluation on those
- Use an index (VS*-tree) over the data signature graph (has light maintenance load) for efficient pruning
1. Encode $Q$ and $G$ to Get Signature Graphs

Query signature graph $Q^*$

```
0100 0000 00010 1000 0000 10000 0000 0100
```

Data signature graph $G^*$

```
0010 1000 00001 1000 0001 0000 0100 10000
```

```
0100 0001 00010 1000 0001 01000 0100 1000 01000
```

```
0000 0100 0000 1000 0000 0010 0100 1000 0001 0100
```

```
0000 0010 0000 1001 00100 0001 0001 0100 01000 1001 1000
```

```
0001 0001 0100 1000 0001 01000
```

```
0000 1000 0000 0010 0000 1000
```

© M. Tamer Ö兹su
ADC PhD School (2015-06-04) 38 / 76
2. Filter-and-Evaluate

Query signature graph $Q^*$

Data signature graph $G^*$

Find matches of $Q^*$ over signature graph $G^*$

Verify each match in RDF graph $G$
How to Generate Candidate List

- Two step process:
  1. For each node of $Q^*$ get lists of nodes in $G^*$ that *include* that node.
  2. Do a multi-way join to get the candidate list.
How to Generate Candidate List

- Two step process:
  1. For each node of $Q^*$ get lists of nodes in $G^*$ that *include* that node.
  2. Do a multi-way join to get the candidate list

- Alternatives:
How to Generate Candidate List

- Two step process:
  1. For each node of $Q^*$ get lists of nodes in $G^*$ that *include* that node.
  2. Do a multi-way join to get the candidate list

- Alternatives:
  - Sequential scan of $G^*$
    - Both steps are inefficient
How to Generate Candidate List

- Two step process:
  1. For each node of $Q^*$ get lists of nodes in $G^*$ that include that node.
  2. Do a multi-way join to get the candidate list

- Alternatives:
  - Sequential scan of $G^*$
    - Both steps are inefficient
  - Use S-trees
    - Height-balanced tree over signatures
    - Run an inclusion query for each node of $Q^*$ and get lists of nodes in $G^*$ that include that node.
      - Given query signature $q$ and a set of data signatures $S$, find all data signatures $s_i \in S$ where $q \& s_i = q$
    - Does not support second step – expensive
How to Generate Candidate List

- Two step process:
  1. For each node of $Q^*$ get lists of nodes in $G^*$ that include that node.
  2. Do a multi-way join to get the candidate list

- Alternatives:
  - Sequential scan of $G^*$
    - Both steps are inefficient
  - Use S-trees
    - Height-balanced tree over signatures
    - Run an inclusion query for each node of $Q^*$ and get lists of nodes in $G^*$ that include that node.
      - Given query signature $q$ and a set of data signatures $S$, find all data signatures $s_i \in S$ where $q \& s_i = q$
    - Does not support second step – expensive
  - VS-tree (and VS*-tree)
    - Multi-resolution summary graph based on S-tree
    - Supports both steps efficiently
    - Grouping by vertices
S-tree Solution

```
0100 0000 00010 1000 0000 10000 0000 0100
```

```
G^1
  d^1_1
  1111 1111
  ^
  0110 1111
      |
G^2
  d^1_1
  0110 1111
  |
G^3
  d^3_1
  0000 1110
  |
  0110 1001
  |
  1100 1001
  |
  1001 1101
  |
  0000 0100
  0100 0001
  1000 0001
  |
  0000 0100
  0010 0000
  1000 0000
```

```
Possibly large join space!
```

© M. Tamer Özsu
ADC PhD School (2015-06-04) 41 / 76
S-tree Solution

The diagram illustrates the S-tree solution with a series of nodes and edges. Each node represents a data point, and the edges show the relationships between them. The diagram is structured into three levels, labeled $G^1$, $G^2$, and $G^3$, with nodes and paths labeled with binary values.

- Level $G^1$: Starting with the root node $0100 0000$, the diagram branches to $1000 0000$ and $0000 0100$.
- Level $G^2$: From $1000 0000$, the diagram continues to $1111 1111$, and from $0000 0100$, it reaches $1101 1101$.
- Level $G^3$: Further branching is seen, with nodes like $0000 1110$, $0110 1111$, and $1100 1001$.

The diagram emphasizes the hierarchical structure and the relationships between different data points.
S-tree Solution

G^1

G^2

G^3
S-tree Solution

$G^1$

$G^2$

$G^3$
S-tree Solution

G¹

G²

G³

Possibly large join space!

© M. Tamer Özsu
ADC PhD School (2015-06-04) 41 / 76
S-tree Solution

Possibly large join space!
Pruning with VS-Tree
Pruning with VS-Tree
Pruning with VS-Tree
Pruning with VS-Tree

G¹

G²

G³

Reduced join space!
Pruning with VS-Tree
Pruning with VS-Tree

Reduced join space!
Applications that rely on RDF data are increasingly popular and are more varied [Verborgh et al., 2014]

Data that are being handled are far more heterogeneous [Duan et al., 2011]

SPARQL queries are becoming more diverse [Arias et al., 2011] and dynamic [Kirchberg et al., 2011]

An experiment [Aluç et al., 2014a]
- No single system is a sole winner across all queries
- No single system is the sole loser across all queries, either
- There can be 2–5 orders of magnitude difference in the performance (i.e., query execution time) between the best and the worst system for a given query
- The winner in one query may timeout in another
- Performance difference widens as dataset size increases
Adaptivity to Workload

- Applications that rely on RDF data are increasingly popular and are more varied [Verborgh et al., 2014]
- Data that are being handled are far more heterogeneous [Duan et al., 2011]
- SPARQL queries are becoming more diverse [Arias et al., 2011] and dynamic [Kirchberg et al., 2011]

Can existing systems cope with these trends – workload diversity & dynamism

No! [Aluç et al., 2014b]

- Fragmented data
- Suboptimal pruning by indexes
- Unnecessarily large sets of intermediate result tuples
Applications that rely on RDF data are increasingly popular and are more varied [Verborgh et al., 2014]

Data that are being handled are far more heterogeneous [Duan et al., 2011]

SPARQL queries are becoming more diverse [Arias et al., 2011] and dynamic [Kirchberg et al., 2011]

An experiment [Aluç et al., 2014a]

No single system is a sole winner across all queries

No single system is a sole loser across all queries, either

There can be 2–5 orders of magnitude difference in the performance (i.e., query execution time) between the best and the worst system for a given query

The winner in one query may timeout in another

Performance difference widens as dataset size increases

---

Our proposal: Idea behind chameleon-db

- When designing and implementing an RDF data management system, assume nothing about the workload upfront
- Organize data dynamically and purely based on the workload

- Performance difference widens as dataset size increases
Group-by-Query Approach

Characteristics:

- Records are not necessarily of fixed length
- Records are not grouped into tables
- Records do not necessarily share the same set of RDF predicates
- Each record represents a very tiny part of the RDF graph
Group-by-Query Approach

Figure: Query 1
Group-by-Query Approach

Figure: Query 1
Group-by-Query Approach

Figure: Query 1
Group-by-Query Approach

Figure: Query 2
Group-by-Query Approach

Figure: Query 2
Group-by-Query Approach

Figure: Query 2
Group-by-Query Approach

Advantages

- Data are physically clustered for the workload
- Better pruning by the indexes
- Fewer intermediate result tuples
Challenges

- **Physical Data Layout**: As the workloads change, the way data are grouped together may no longer be suitable
  - Hierarchical Clustering Algorithm [Aluç et al., 2015]
  - Tunable-LSH [Aluç et al., 2015]

- **Indexing**: Indexing upfront is not a choice

- **Query Evaluation**: Can we execute queries efficiently even when the physical layout is constantly changing? [Aluç et al., 2015]
Prototype system [Aluç et al., 2013]
35,000 lines of code in C++ under Linux (plus code for SPARQL 1.0 parser)
Some Open Problems

- Scalability of the solutions to very large datasets
- Maintenance of auxiliary data structures in dynamic environments
- Adaptive systems to handle varying and time-changing workloads
- Uncertain RDF data processing
- Keyword search over RDF data
- Query processing over incomplete RDF data
Outline

1. Introduction – Graph Types
2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics
3. RDF Graph Querying
   - Data Warehousing
   - Distributed SPARQL Execution
   - Linked Object Data Querying
Remember the Environment

- Distributed environment
- Some of the data sites can process SPARQL queries – SPARQL endpoints
- Not all data sites can process queries
Remember the Environment

- Distributed environment
- Some of the data sites can process SPARQL queries – SPARQL endpoints
- Not all data sites can process queries
- Alternatives
Remember the Environment

- Distributed environment
- Some of the data sites can process SPARQL queries – SPARQL endpoints
- Not all data sites can process queries
- Alternatives
  - Data re-distribution + query decomposition
Remember the Environment

- Distributed environment
- Some of the data sites can process SPARQL queries – SPARQL endpoints
- Not all data sites can process queries
- Alternatives
  - Data re-distribution + query decomposition
  - SPARQL federation: just process at SPARQL endpoints
Remember the Environment

- Distributed environment
- Some of the data sites can process SPARQL queries – SPARQL endpoints
- Not all data sites can process queries
- Alternatives
  - Data re-distribution + query decomposition
  - SPARQL federation: just process at SPARQL endpoints
  - Live querying (see next section)
Data partitioning approaches

- RDF data warehouse is partitioned and distributed
  - RDF data $D = \{D_1, \ldots, D_n\}$
  - Allocate each $D_i$ to a site
Data partitioning approaches

- RDF data warehouse is partitioned and distributed
  - RDF data $D = \{D_1, \ldots, D_n\}$
  - Allocate each $D_i$ to a site

Partitioning alternatives

- Table-based (e.g., [Husain et al., 2011])
- Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
- Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])
Data partitioning approaches

- RDF data warehouse is partitioned and distributed
  - RDF data $D = \{D_1, \ldots, D_n\}$
  - Allocate each $D_i$ to a site

- Partitioning alternatives
  - Table-based (e.g., [Husain et al., 2011])
  - Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
  - Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])

- SPARQL query decomposed $Q = \{Q_1, \ldots, Q_k\}$
- Distributed execution of $\{Q_1, \ldots, Q_k\}$ over $\{D_1, \ldots, D_n\}$

▶ High performance
▶ Great for parallelizing centralized RDF data
▶ May not be possible to re-partition and re-allocate Web data (i.e., LOD)
Distributed RDF Processing

[Kaoudi and Manolescu, 2015]

Data partitioning approaches

- RDF data warehouse is partitioned and distributed
  - RDF data $D = \{ D_1, \ldots, D_n \}$
  - Allocate each $D_i$ to a site

- Partitioning alternatives
  - Table-based (e.g., [Husain et al., 2011])
  - Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
  - Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])

- SPARQL query decomposed $Q = \{ Q_1, \ldots, Q_k \}$
- Distributed execution of $\{ Q_1, \ldots, Q_k \}$ over $\{ D_1, \ldots, D_n \}$

- High performance
- Great for parallelizing centralized RDF data
- May not be possible to re-partition and re-allocate Web data (i.e., LOD)
Data summary-based approaches

- Build summaries (index) for the distributed RDF datasets (e.g., [Atre et al., 2010; Prasser et al., 2012])
Data summary-based approaches

- Build summaries (index) for the distributed RDF datasets (e.g., [Atre et al., 2010; Prasser et al., 2012])
- SPARQL query $Q = \{Q_1, \ldots, Q_k\}$
- Distributed execution of $\{Q_1, \ldots, Q_k\}$ using the data summary
Data summary-based approaches

- Build summaries (index) for the distributed RDF datasets (e.g., [Atre et al., 2010; Prasser et al., 2012])
- SPARQL query $Q = \{Q_1, \ldots, Q_k\}$
- Distributed execution of $\{Q_1, \ldots, Q_k\}$ using the data summary

- No data re-partitioning and re-allocation
- Have to scan the data at each site
- Index over distributed data with maintenance concerns
Consider only the SPARQL endpoints for query execution
No data re-partitioning/re-distribution
Consider $D = D_1 \cup D_2 \cup \ldots \cup D_n$; $D_i$: SPARQL endpoint
Consider only the SPARQL endpoints for query execution
No data re-partitioning/re-distribution
Consider \( D = D_1 \cup D_2 \cup \ldots \cup D_n \); \( D_i \): SPARQL endpoint

Alternatives
- SPARQL query decomposed \( Q = \{Q_1, \ldots, Q_k\} \) and executed over \( \{D_1, \ldots, D_n\} \) – DARQ, FedX [Schwarte et al., 2011], SPLENDID [Görlitz and Staab, 2011], ANAPSID [Acosta et al., 2011]
- Partial query evaluation – Distributed gStore [Peng et al., 2014]
Consider only the SPARQL endpoints for query execution
No data re-partitioning/re-distribution
Consider $D = D_1 \cup D_2 \cup \ldots \cup D_n$; $D_i$: SPARQL endpoint
Alternatives
- SPARQL query decomposed $Q = \{Q_1, \ldots, Q_k\}$ and executed over \{\$D_1, \ldots, D_n\} – DARQ, FedEx [Schwarte et al., 2011], SPLENDID [Görlitz and Staab, 2011], ANAPSID [Acosta et al., 2011]
- Partial query evaluation – Distributed gStore [Peng et al., 2014]

Partial evaluation
- Given function $f(s, d)$ and part of its input $s$, perform $f$’s computation that only depends on $s$ to get $f'(d)$
- Compute $f'(d)$ when $d$ becomes available
- Applied to, e.g., XML [Buneman et al., 2006]
Distributed SPARQL Using Partial Query Evaluation

Two steps:

1. Evaluate a query at each site to find local matches
   - Query is the function and each $D_i$ is the known input
   - Inner match or local partial match

- Crossing match
- Centralized assembly
- Distributed assembly

© M. Tamer Özsu
Distributed SPARQL Using Partial Query Evaluation

Two steps:
1. Evaluate a query at each site to find local matches
   - Query is the function and each $D_i$ is the known input
   - Inner match or local partial match
2. Assemble the partial matches to get final result
   - Crossing match
   - Centralized assembly
   - Distributed assembly
Some Open Problems

- Handling data at non-SPARQL endpoint sites
- Modification to SPARQL endpoints (for partial query evaluation)
- Heterogeneous use of vocabularies (use of ontologies)
Outline

1 Introduction – Graph Types

2 Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics

3 RDF Graph Querying
   - Data Warehousing
   - Distributed SPARQL Execution
   - Linked Object Data Querying
Closer Look
Globally Distributed Network of Data
Traditional Hypertext-based Web Access

Data exposed to the Web via HTML
Linked Data Publishing Principles

Data model: RDF
Global identifier: URI
Access mechanism: HTTP
Connection: data links
Live Query Processing

- Not all data resides at SPARQL endpoints
- Freshness of access to data important
- Potentially countably infinite data sources
- Live querying
  - On-line execution
  - Only rely on linked data principles
- Alternatives
  - Traversal-based approaches
  - Index-based approaches
  - Hybrid approaches
Web Document

Given a countably infinite set $\mathcal{D}$ (documents), a Web of Linked Data is a tuple $W = (\mathcal{D}, \text{adoc}, \text{data})$ where:

- $\mathcal{D} \subseteq \mathcal{D}$,
- $\text{adoc}$ is a partial mapping from URIs to $\mathcal{D}$, and
- $\text{data}$ is a total mapping from $\mathcal{D}$ to finite sets of RDF triples.
Linked Data Model

**Web Document**

Given a countably infinite set $D$ (documents), a Web of Linked Data is a tuple $W = (D, adoc, data)$ where:

- $D \subseteq \mathcal{D}$,
- $adoc$ is a partial mapping from URIs to $D$, and
- $data$ is a total mapping from $D$ to finite sets of RDF triples.

**Web of Linked Data**

A Web of Linked Data $W = (D, adoc, data)$ contains a data link from document $d \in D$ to document $d' \in D$ if there exists a URI $u$ such that:

- $u$ is mentioned in an RDF triple $t \in data(d)$, and
- $d' = adoc(u)$. 
Full-web semantics

- Scope of evaluating a SPARQL expression is all Linked Data
- Query result completeness cannot be guaranteed by any (terminating) execution
SPARQL Query Semantics in Live Querying

- **Full-web semantics**
  - Scope of evaluating a SPARQL expression is **all** Linked Data
  - Query result completeness cannot be guaranteed by any (terminating) execution

- **Reachability-based query semantics**
  - Query consists of a SPARQL expression, a set of seed URIs $S$, and a reachability condition $c$
  - Scope: all data along paths of data links that satisfy the condition
  - Computationally feasible
Traversing Approaches

- Discover relevant URIs recursively by traversing (specific) data links at query execution runtime [Hartig, 2013; Ladwig and Tran, 2011]
- Implements reachability-based query semantics
  - Start from a set of seed URIs
  - Recursively follow and discover new URIs
- Important issue is selection of seed URIs
- Retrieved data serves to discover new URIs and to construct result
Traversing Approaches

- Discover relevant URIs recursively by traversing (specific) data links at query execution runtime [Hartig, 2013; Ladwig and Tran, 2011]
- Implements reachability-based query semantics

Advantages

- Easy to implement.
- No data structure to maintain.
- Important issue is selection of seed URIs
- Retrieved data serves to discover new URIs and to construct result
Traversing Approaches

- Discover relevant URIs recursively by traversing (specific) data links at query execution runtime [Hartig, 2013; Ladwig and Tran, 2011]
- Implements reachability-based query semantics

Start from a set of seed URIs
Recursively follow and discover new URIs

Important issue is selection of seed URIs
Retrieved data serves to discover new URIs and to construct result

Advantages

Easy to implement.
No data structure to maintain.

Disadvantages

Possibilities for parallelized data retrieval are limited
Repeated data retrieval introduces significant query latency.
Index Approaches

- Use pre-populated index to determine relevant URIs (and to avoid as many irrelevant ones as possible)
- Different index keys possible; e.g., triple patterns [Umbrich et al., 2011]
  - Index entries a set of URIs
  - Indexed URIs may appear multiple times (i.e., associated with multiple index keys)
  - Each URI in such an entry may be paired with a cardinality (utilized for source ranking)
Index Approaches

- Use pre-populated index to determine relevant URIs (and to avoid as many irrelevant ones as possible)
- Different index keys possible; e.g., triple patterns [Umbrich et al., 2011]
  - Index entries a set of URIs

Advantages

Data retrieval can be fully parallelized
Reduces the impact of data retrieval on query execution time
Index Approaches

- Use pre-populated index to determine relevant URIs (and to avoid as many irrelevant ones as possible)
- Different index keys possible; e.g., triple patterns [Umbrich et al., 2011]
  - Index entries a set of URIs

**Advantages**

Data retrieval can be fully parallelized
Reduces the impact of data retrieval on query execution time

**Disadvantages**

Querying can only start after index construction
Depends on what has been selected for the index
Freshness may be an issue
Index maintenance
Hybrid Approach

- Perform a traversal-based execution using a prioritized list of URIs to look up [Ladwig and Tran, 2010]
- Initial seed from the pre-populated index
- Non-seed URIs are ranked by a function based on information in the index
- New discovered URIs that are not in the index are ranked according to number of referring documents
Some Open Problems

- Optimize queries by using statistics collected during earlier query executions
- Heterogeneous use of vocabularies (use of ontologies)
- Combine SPARQL federation to leverage SPARQL endpoint functionality
This presentation draws upon collaborative research and discussions with the following colleagues (in alphabetical order)

Güneş Aluç, U. Waterloo

Khaled Ammar, U. Waterloo

Khuzaima Daudjee, U. Waterloo

Young Han, U. Waterloo

Olaf Hartig, U. Waterloo

Lei Chen, Hong Kong UST

Lei Zou, Peking Univ.
Thank you!

Research supported by

NSERC

CRSNG

INNOVATION.CA

Ontario

CAS Research
Technology Incubation Lab

IBM Canada Lab


