TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing

Sairam Gurajada†, Stephan Seufert†, Iris Miliaraki†, Martin Theobald‡

†Databases & Information Systems Group
Max-Planck Institute for Informatics
Saarbrücken, Germany

‡ ADReM Research Group
University of Antwerp
Antwerp, Belgium
Resource Description Framework (RDF)

RDF is a data model for representing information on the Web
RDF is a data model for representing information on the Web.

Example datasets: YAGO (>120 M facts), DBpedia (>400 M facts), Freebase, ...

RDF triples obtained from IE

Barack Obama isA President_of_USA.
Barack Obama bornIn Honolulu.
Barack Obama won Nobel_Peace_Prize.
Barack Obama memberOf Democratic_Party.

Barack Obama

From Wikipedia, the free encyclopedia

"Obama" redirects here. For other uses, see Obama (disambiguation).

This article is about the 44th president of the United States. For his father, see Barack Obama, Sr.

Barack Hussein Obama II (born August 4, 1961) is the 44th and current President of the United States, and the first African American to hold the office. Born in Honolulu, Hawaii, Obama is a graduate of Columbia University and Harvard Law School, where he served as president of the Harvard Law Review. He was a community organizer in Chicago before earning his law degree. He worked as a civil rights attorney and taught constitutional law at the University of Chicago Law School from 1992 to 2004. He served three terms representing the 13th District in the Illinois Senate from 1997 to 2004, running unsuccessfully for the United States House of Representatives in 2000.

In 2004, Obama received national attention during his campaign to represent Illinois in the United States Senate with his victory in the March Democratic primary, his keynote address at the Democratic National Convention in July, and his election to the Senate in November. He began his presidential campaign in 2007 and, after a close primary campaign against Hillary Rodham Clinton in 2008, he won sufficient delegates in the Democratic Party primaries to receive the presidential nomination. He then defeated Republican nominee John McCain in the general election, and was inaugurated as president on January 20, 2009. Nine months after his election, Obama was named the 2009 Nobel Peace Prize laureate.

During his first two years in office, Obama signed into law economic stimulus legislation in response to the Great Recession in the form of the American Recovery and Reinvestment Act of 2009 and the Tax Relief, Unemployment Insurance Reauthorization, and Job Creation Act of 2010. Other major domestic initiatives in his first term included the Patient Protection and Affordable Care Act, often referred to as "Obamacare"...
RDF is a data model for representing information on the Web.

RDF triples obtained from IE:
- Barack Obama isA President_of_USA.
- Barack Obama bornIn Honolulu.
- Barack Obama won Nobel_Peace_Prize.
- Barack Obama memberOf Democratic_Party.

Example datasets:
- YAGO (≈ 120 M facts)
- DBpedia (≈ 400 M facts)
- Freebase, ...
RDF is a data model for representing information on the Web

Example datasets:
- YAGO (>120 M facts)
- DBpedia (>400 M facts)
- Freebase, ...

**RDF triples obtained from IE**

- Barack Obama isA President_of_USA.
- Barack Obama bornIn Honolulu.
- Barack Obama won Nobel_Peace_Prize.
- Barack Obama memOf Democratic_Party.

**Barack Obama**
- bornIn Honolulu
- memOf Democratic_Party

**Lady Gaga**
- Singer
- memOf Grammy_Award
- bornIn New_York

**Barack Obama**
- bornIn New_Haven
- memOf Republican_Party

**Jimmy Carter**
- bornIn Plains
- memOf Republican_Party

**George W Bush**
- bornIn Texas
- memOf Republican_Party

**Texas**
- capitol
- isA state

**New York**
- state
- isA state

**New Haven**
- city
- isA city

**Honolulu**
- city
- isA city

**USA**
- country
- isA country

**Plains**
- county
- isA county

**Nobel_Peace_Prize**
- award
- isA award

**Grammy_Award**
- award
- isA award

**Republican_Party**
- political
- isA political

**Democratic_Party**
- political
- isA political

**Singer**
- occupation
- isA occupation

**President_of_USA**
- occupation
- isA occupation

**isA**
- type
- isA type

**bornIn**
- location
- isA location

**won**
- action
- isA action

**memOf**
- membership
- isA membership
As of 2011, there exists more than 30 billion triples in the linked data cloud (from more than 300 sources).


Source: http://linkeddata.org
Indexing and Querying RDF Data

- RDF triples are stored and indexed in a relational table (relational approach)
- SPARQL is the language suggested by W3C for querying RDF data
- SPARQL has many similarities with standard SQL
- **SELECT-PROJECT-JOIN** forms the main building blocks of SPARQL
Indexing and Querying RDF Data

- RDF triples are stored and indexed in a relational table (relational approach)
- SPARQL is the language suggested by W3C for querying RDF data
- SPARQL has many similarities with standard SQL
- `SELECT-PROJECT-JOIN` forms the main building blocks of SPARQL

**SPARQL Query:**

Find persons who are born in USA and won a prize.

```sparql
SELECT ?person, ?city, ?prize WHERE {
  (R1) ?person <bornIn> ?city .
  (R2) ?city <locatedIn> USA .
  (R3) ?person <won> ?prize . }
```

**RDF Data:**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barack Obama</td>
<td>bornIn</td>
<td>Honolulu</td>
</tr>
<tr>
<td>Barack Obama</td>
<td>won</td>
<td>Nobel Peace Prize</td>
</tr>
<tr>
<td>Barack Obama</td>
<td>won</td>
<td>Grammy Award</td>
</tr>
<tr>
<td>Honolulu</td>
<td>locatedIn</td>
<td>United States</td>
</tr>
<tr>
<td>Barack Obama</td>
<td>memberOf</td>
<td>Republican Party</td>
</tr>
<tr>
<td>John F. Kennedy</td>
<td>bornIn</td>
<td>United States</td>
</tr>
<tr>
<td>John F. Kennedy</td>
<td>memberOf</td>
<td>Republican Party</td>
</tr>
<tr>
<td>John F. Kennedy</td>
<td>diedIn</td>
<td>Dallas</td>
</tr>
<tr>
<td>Dallas</td>
<td>locatedIn</td>
<td>United States</td>
</tr>
<tr>
<td>Brookline</td>
<td>locatedIn</td>
<td>United States</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\[ ?person \xrightarrow{bornIn} ?city \xrightarrow{locatedIn} USA \]

\[ ?person \xrightarrow{won} ?prize \]
Indexing and Querying RDF Data

- RDF triples are stored and indexed in a relational table (relational approach)
- SPARQL is the language suggested by W3C for querying RDF data
- SPARQL has many similarities with standard SQL
- SELECT-PROJECT-JOIN forms the main building blocks of SPARQL

**SPARQL Query:**

Find persons who are born in USA and won a prize.

```
SELECT ?person, ?city, ?prize WHERE {
  (R1) ?person <bornIn> ?city .
  (R2) ?city <locatedIn> USA .
  (R3) ?person <won> ?prize .
}
```

**RDF Data:**

```
<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barack Obama</td>
<td>bornIn</td>
<td>Honolulu</td>
</tr>
<tr>
<td>Barack Obama</td>
<td>won</td>
<td>Nobel Peace Prize</td>
</tr>
<tr>
<td>Barack Obama</td>
<td>locatedIn</td>
<td>Republican Party</td>
</tr>
<tr>
<td>Honolulu</td>
<td>memberOf</td>
<td>United States</td>
</tr>
<tr>
<td>Barack Obama</td>
<td>won</td>
<td>Grammy Award</td>
</tr>
<tr>
<td>John F. Kennedy</td>
<td>bornIn</td>
<td>United States</td>
</tr>
<tr>
<td>John F. Kennedy</td>
<td>memberOf</td>
<td>Republican Party</td>
</tr>
<tr>
<td>John F. Kennedy</td>
<td>diedIn</td>
<td>Dallas</td>
</tr>
<tr>
<td>Dallas</td>
<td>locatedIn</td>
<td>United States</td>
</tr>
<tr>
<td>Brookline</td>
<td>locatedIn</td>
<td>United States</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

**Results:** (R1 ⋈ R2 ⋈ R3)

```
<table>
<thead>
<tr>
<th>?person</th>
<th>?city</th>
<th>?prize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barack Obama</td>
<td>Honolulu</td>
<td>Peace_Nobel_Place</td>
</tr>
<tr>
<td>Barack Obama</td>
<td>Honolulu</td>
<td>Grammy_Award</td>
</tr>
<tr>
<td>Barack Obama</td>
<td>Honolulu</td>
<td>...</td>
</tr>
</tbody>
</table>
```

...
Current Approaches

**Efficiency** – thoroughly investigated in single-node setting

- crucial factors – *Join-order optimization, Join-ahead pruning, indexing layout, choice of operators*
- Eg. Jena, Sesame, HexaStore, MonetDB-RDF, SW-Store, RDF-3X, TripleBit, BitMat, gStore, ...
Current Approaches

**Efficiency** – thoroughly investigated in single-node setting
- crucial factors – *Join-order optimization, Join-ahead pruning, indexing layout, choice of operators*
- Eg. Jena, Sesame, HexaStore, MonetDB-RDF, SW-Store, RDF-3X, TripleBit, BitMat, gStore, ...

**Scalability** – recently a line of distributed systems has been proposed
- SHARD, H-RDF-3X, Trinity.RDF, ...

Relational-based  Graph-based
Current Approaches

**Efficiency** – thoroughly investigated in single-node setting
- crucial factors – *Join-order optimization, Join-ahead pruning, indexing layout, choice of operators*
- Eg. Jena, Sesame, HexaStore, MonetDB-RDF, SW-Store, RDF-3X, TripleBit, BitMat, gStore, ...

**Scalability** – recently a line of distributed systems has been proposed
- SHARD, H-RDF-3X, Trinity.RDF, ...
  - Relational-based
  - Graph-based

**Relational-based (Joins) vs Graph-based (Exploration) [distributed setting]**
- SPARQL 1.0 requires a row-oriented output → *joins are inevitable*
- Relational approaches suffer from “large intermediate relations”
  - (inaccurate/insufficient statistics resulting in poor query plans)
Current Approaches

**Efficiency** – thoroughly investigated in single-node setting
- crucial factors – *Join-order optimization, Join-ahead pruning, indexing layout, choice of operators*
- Eg. Jena, Sesame, HexaStore, MonetDB-RDF, SW-Store, RDF-3X, TripleBit, BitMat, gStore, ...

**Scalability** – recently a line of distributed systems has been proposed
- SHARD, H-RDF-3X, Trinity.RDF, ...
  - **Relational-based**
  - **Graph-based**

**Relational-based (Joins) vs Graph-based (Exploration) [distributed setting]**
- SPARQL 1.0 requires a row-oriented output → joins are inevitable
- Relational approaches suffer from “large intermediate relations”
  (inaccurate/insufficient statistics resulting in poor query plans)
- Graph-based systems use **distributed graph exploration** followed by relational joins
- Effective when graph exploration prunes a lot of bindings
  (in case of selective queries)
Current Approaches

**Efficiency** – thoroughly investigated in single-node setting
- crucial factors – *Join-order optimization, Join-ahead pruning, indexing layout, choice of operators*
- Eg. Jena, Sesame, HexaStore, MonetDB-RDF, SW-Store, RDF-3X, TripleBit, BitMat, gStore, ...

**Scalability** – recently a line of distributed systems has been proposed
- `SHARD, H-RDF-3X, Trinity.RDF, ...`
  - Relational-based
  - Graph-based

**Relational-based (Joins) vs Graph-based (Exploration) [distributed setting]**
- SPARQL 1.0 requires a row-oriented output → joins are inevitable
- Relational approaches suffer from “large intermediate relations” (inaccurate/insufficient statistics resulting in poor query plans)
- Graph-based systems use *distributed graph exploration* followed by relational joins
- Effective when graph exploration prunes a lot of bindings (in case of selective queries)

**Problems with existing distributed relational-based RDF engines**
- 1. Synchronous processing of joins
- 2. Dangling triples – occur in intermediate relations but not in final results
1. Synchronous Processing of Joins

Consider a query with four relations $R_1, R_2, R_3, R_4$ with join order:

$$(R_1 \bowtie_2 R_2) \bowtie_1 (R_3 \bowtie_3 R_4)$$

Our approach: Minimize dependencies among join operators and only synchronize when needed! (using asynchronous communication (MPICH2) protocol)
1. Synchronous Processing of Joins

Consider a query with four relations $R_1, R_2, R_3, R_4$ with join order:

$$(R_1 \bowtie_2 R_2) \bowtie_1 (R_3 \bowtie_3 R_4)$$

**Synchronous case 1:**
$\text{MR\_job1} \rightarrow \text{MR\_job2} \rightarrow \text{MR\_job3}$
1. Synchronous Processing of Joins

Consider a query with four relations $R_1$, $R_2$, $R_3$, $R_4$ with join order:

$$(R_1 \bowtie_2 R_2) \bowtie_1 (R_3 \bowtie_3 R_4)$$

**Synchronous case 1:**
MR_job1 $\rightarrow$ MR_job2 $\rightarrow$ MR_job3

**Synchronous case 2:**
MR_job1 $\rightarrow$ MR_job2

Our approach: Minimize dependancies among join operators and only synchronize when needed! (using asynchronous communication (MPICH2) protocol)
1. Synchronous Processing of Joins

Consider a query with four relations $R_1, R_2, R_3, R_4$ with join order:

$$(R_1 \bowtie_2 R_2) \bowtie_1 (R_3 \bowtie_3 R_4)$$

**Synchronous case 1:**
$\text{MR}_\text{job}1 \rightarrow \text{MR}_\text{job}2 \rightarrow \text{MR}_\text{job}3$

**Synchronous case 2:**
$\text{MR}_\text{job}1 \rightarrow \text{MR}_\text{job}2$

**Synchronous case 3:**
$(\text{MR}_\text{job}1 \parallel \text{MR}_\text{job}2) \rightarrow \text{MR}_\text{job}3$

Our approach: Minimize dependencies among join operators and only synchronize when needed! (using asynchronous communication (MPICH2) protocol)
1. Synchronous Processing of Joins

Consider a query with four relations $R_1$, $R_2$, $R_3$, $R_4$ with join order:

$$(R_1 \bowtie_2 R_2) \bowtie_1 (R_3 \bowtie_3 R_4)$$

**Synchronous case 1:**
MR_job1 $\rightarrow$ MR_job2 $\rightarrow$ MR_job3

**Synchronous case 2:**
MR_job1 $\rightarrow$ MR_job2

**Synchronous case 3:**
(MR_job1 $|$ MR_job2) $\rightarrow$ MR_job3

Our approach: Minimize dependancies among join operators and only synchronize when needed! (using asynchronous communication (MPICH2) protocol)
2. Pruning Dangling Triples

Consider a query with four relations $R_1$, $R_2$, $R_3$, $R_4$ with join order:

$$(R_1 \bowtie_2 R_2) \bowtie_1 (R_3 \bowtie_3 R_4)$$

---

1RDF-3X: a RISC-style Engine for RDF, VLDBJ 2010
2. Pruning Dangling Triples

Consider a query with four relations $R_1, R_2, R_3, R_4$ with join order:

\[(R_1 \bowtie_2 R_2) \bowtie_1 (R_3 \bowtie_3 R_4)\]

**Sideways Information Passing (SIP) of RDF-3X**
- Powerful runtime pruning technique
- Shares information across join operators
- Requires synchronization

---

1RDF-3X: a RISC-style Engine for RDF, VLDBJ 2010
2. Pruning Dangling Triples

Consider a query with four relations $R_1$, $R_2$, $R_3$, $R_4$ with join order:

$$(R_1 \bowtie_2 R_2) \bowtie_1 (R_3 \bowtie_3 R_4)$$

**Sideways Information Passing (SIP) of RDF-3X**

- Powerful runtime pruning technique
- Shares information across join operators
- Requires synchronization

**Our approach:** Join-ahead pruning

1. Pre-partition the triples (into groups)

---

1. RDF-3X: a RISC-style Engine for RDF, VLDBJ 2010
2. Pruning Dangling Triples

Consider a query with four relations $R_1, R_2, R_3, R_4$ with join order:

$$(R_1 \bowtie_2 R_2) \bowtie_1 (R_3 \bowtie_3 R_4)$$

**Sideways Information Passing (SIP) of RDF-3X**

- Powerful runtime pruning technique
- Shares information across join operators
- Requires synchronization

**Our approach:** Join-ahead pruning

1. Pre-partition the triples (into groups)
2. Query over groups to find the ones which are relevant to the query

---

1RDF-3X: a RISC-style Engine for RDF, VLDBJ 2010
2. Pruning Dangling Triples

Consider a query with four relations $R_1$, $R_2$, $R_3$, $R_4$ with join order:

$$(R_1 \bowtie_2 R_2) \bowtie_1 (R_3 \bowtie_3 R_4)$$

**Sideways Information Passing (SIP) of RDF-3X**

- Powerful runtime pruning technique
- Shares information across join operators
- Requires synchronization

**Our approach: Join-ahead pruning**

1. Pre-partition the triples (into groups)
2. Query over groups to find the ones which are relevant to the query
3. Scan & Join only triples from the relevant groups

---

1RDF-3X: a RISC-style Engine for RDF, VLDBJ 2010
TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing

2. Join-ahead Pruning via Graph Summarization
2. Join-ahead Pruning via Graph Summarization

Locality-based grouping (using METIS)
2. Join-ahead Pruning via Graph Summarization

Locality-based grouping (using METIS)

Summary Graph

SPARQL Query:
```
SELECT ?city, ?prize WHERE {
  Barack Obama <bornIn> ?city .
  ?city <locatedIn> USA .
  Barack Obama <won> ?prize .
}
```

Supernode Bindings:
```
?city : P1
?prize : P2, P4
```

SPARQL Query:
```
SELECT ?city, ?prize WHERE {
  Barack Obama <bornIn> ?city .
  ?city <locatedIn> USA .
}
```

Supernode Bindings:
```
!!! Empty Result
```

False positives may occur but no false negatives!!!
2. Join-ahead Pruning via Graph Summarization

Locality-based grouping (using METIS)

Summary Graph

SPARQL Query:
SELECT ?city, ?prize WHERE {
  Barack Obama <bornIn> ?city .
  ?city <locatedIn> USA .
  Barack Obama <won> ?prize .
}

Supernode Bindings:
?city : P1
?prize : P2, P4

False positives may occur but no false negatives!!!
2. Join-ahead Pruning via Graph Summarization

Locality-based grouping (using METIS)

[Diagram of locality-based grouping]

SPARQL Query:
SELECT ?city, ?prize WHERE {
Barack Obama <bornIn> ?city .
?city <locatedIn> USA .
}

Supernode Bindings: !!! Empty Result
?city : --
?prize : --
2. Join-ahead Pruning via Graph Summarization

**Locality-based grouping (using METIS)**

**Summary Graph**

**SPARQL Query:**
SELECT ?city, ?prize WHERE {
  Barack Obama <bornIn> ?city .
  ?city <locatedIn> USA .
}

**Supernode Bindings:** !!! Empty Result

?city : --
?prize : --

False positives may occur but no false negatives!!!
TriAD (for “Triple Asynchronous and Distributed”) – a distributed RDF Store
TriAD (for “Triple Asynchronous and Distributed”) – a distributed RDF Store

Main-memory backed six permutation distributed indexes (with locality-aware sharding and encoded summary information)
TriAD Distributed RDF Store

TriAD (for “Triple Asynchronous and Distributed”) – a distributed RDF Store

- Asynchronous Processing of Joins via MPICH2 communication protocol
- Main-memory backed six permutation distributed indexes (with locality-aware sharding and encoded summary information)
TriAD (for “Triple Asynchronous and Distributed”) – a distributed RDF Store

- Join-ahead pruning via Graph Summarization
- Asynchronous Processing of Joins via MPICH2 communication protocol
- Main-memory backed six permutation distributed indexes (with locality-aware sharding and encoded summary information)
TriAD (for “Triple Asynchronous and Distributed”) – a distributed RDF Store

- **Two-stage Query Optimization**
  - (distributed, multi-threading, and join-ahead pruning into account)

- **Join-ahead pruning** via Graph Summarization

- **Asynchronous Processing of Joins** via MPICH2 communication protocol

- **Main-memory backed six permutation distributed indexes**
  - (with locality-aware sharding and encoded summary information)
TriAD Architecture

RDF Data  METIS Partitions Info  Master Node  SPARQL Query

RDF Parser  Summary Triples  Global Statistics  Query Optimizer  SPARQL Query Parser

Bidirectional Dictionaries  Data Triples  Global Statistics  Local Statistics  Global Query Plan  Results

Local Query Processor

Local SPO Indexes

MPICH2 - Asynchronous Communication Protocol

Encoded Triples  Intermediate Results  Encoded Triples  Intermediate Results  Encoded Triples  Intermediate Results

Local Query Processor

Local SPO Indexes

Slave 1  Slave 2  Slave n

SPO  SOP  PSO  OSP  OPS  POS
Indexing

1. RDF Parser
   - RDF Data
   - METIS Partitions Info
   - Summary Triples
   - Data Triples
   - Bidirectional Dictionaries
   - Partitioner

Master Node
- Global Statistics
- Query Optimizer
- SPARQL Query
- Summary Graph
- Supernodes
- Query Graph
- Local Statistics
- Global Query Plan
- Results

Local Query Processor
- Local SPO Indexes
- Slave 1
- Slave 2
- Slave n

RDF Data
- RDF Parser
- Query
- Results

MPICH2 - Asynchronous Communication Protocol
- Encoded Triples
- Query Plan
- Intermediate Results

Horizontal Partitioning

Statistics
- Local
- Global

Summary Triples
- Local Query Processor
- Local SPO Indexes
- Slave 1
- Slave 2
- Slave n

Supernodes
- Local Query Processor
- Local SPO Indexes
- Slave 1
- Slave 2
- Slave n

Query Graph
- Local Query Processor
- Local SPO Indexes
- Slave 1
- Slave 2
- Slave n

Intermediate Results
- Local Query Processor
- Local SPO Indexes
- Slave 1
- Slave 2
- Slave n

Encoded Triples
- Local Query Processor
- Local SPO Indexes
- Slave 1
- Slave 2
- Slave n

Query Plan
- Local Query Processor
- Local SPO Indexes
- Slave 1
- Slave 2
- Slave n

Intermediate Results
- Local Query Processor
- Local SPO Indexes
- Slave 1
- Slave 2
- Slave n

Local SPO Indexes
- Local Query Processor
- Slave 1
- Slave 2
- Slave n

Slave 1
- Local Query Processor
- Local SPO Indexes
- Slave 1
- Slave 2
- Slave n

Slave 2
- Local Query Processor
- Local SPO Indexes
- Slave 1
- Slave 2
- Slave n

Slave n
- Local Query Processor
- Local SPO Indexes
- Slave 1
- Slave 2
- Slave n

TriAD Architecture
TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing

TriAD Architecture

Indexing

MPICH2 - Asynchronous Communication Protocol

Local Query Processor
Local SPO Indexes

Slave 1

SPO | SOP | PSO | OSP | OPS | POS
---|----|----|----|----|----

Slave 2

SPO | SOP | PSO | OSP | OPS | POS
---|----|----|----|----|----

Slave n

SPO | SOP | PSO | OSP | OPS | POS
---|----|----|----|----|----
TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing

TriAD Architecture

Indexing

1. RDF Parser
2. Bidirectional Dictionaries
3. Partitioner
4. Global Statistics

MPICH2 - Asynchronous Communication Protocol

1. RDF Data
2. METIS Partitions Info
3. Master Node

SPARQL Query Processing

1. SPARQL Query Parser
2. Query Optimizer
3. SPARQL Results

Local Query Processors

1. Slave 1
2. Slave 2
3. Slave n

Local SPO Indexes

1. SPO
2. SOP
3. PSO
4. OSP
5. OPS
6. POS
TriAD Architecture

SPARQL Query Processing

Stage 1

**RDF Data**
- METIS Partitions Info
- Summary Triples
- Data Triples
- Horizontal Partitioning
- Bidirectional Dictionaries
- Partitioner

**SPARQL Query**
- Query Parser
- Query Graph
- Supernodes

**Master Node**
- Summary Graph
- Local Statistics
- Global Query Plan

**Local Query Processor**
- Local SPO Indexes

**Encoded Triples**
- Query Plan
- Intermediate Results

**MPICH2 - Asynchronous Communication Protocol**

**Slave 1**
- Local Query Processor

**Slave 2**
- Local Query Processor

**Slave n**
- Local Query Processor

**RDF Data**
- RDF Parser
- Statistics
- RDF Data METIS Partitions Info

**Statistics**
- Local Results
- Global Results

**Global Statistics**
- Query Optimizer
- SPARQL Results

**Global Query Plan**
- Results
TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing

TriAD Architecture

SPARQL Query Processing

Stage 1

Stage 2
Indexing

RDF graph partitioning

- using a locality-based non-overlapping partitioning algorithm
- i.e. Each \( s \) (or \( o \)) of RDF triple \( \langle s, p, o \rangle \) mapped to one supernode \( P_s \) (or \( P_o \))
Indexing

RDF graph partitioning
- using a locality-based non-overlapping partitioning algorithm
- i.e. Each $s$ (or $o$) of RDF triple $\langle s, p, o \rangle$ mapped to one supernode $P_s$ (or $P_o$)

Triple encoding
- Each triple $\langle s, p, o \rangle$ is encoded into two triples: data triple, summary triple
- **Dictionary encoding:** Each entity is assigned a globally unique id is computed by concatenating (supernode id) $P_s$ and a (local id) $id_s$
  Eg. $\langle\text{Barack Obama}, \text{won}, \text{Nobel Prize}\rangle$
  - Data triple: $\langle 1||1,6,4||3 \rangle$  Summary triple $\langle 1,6,4 \rangle$
Indexing

RDF graph partitioning
- using a locality-based non-overlapping partitioning algorithm
- i.e. Each s (or o) of RDF triple \( \langle s, p, o \rangle \) mapped to one supernode \( P_s \) (or \( P_o \))

Triple encoding
- Each triple \( \langle s, p, o \rangle \) is encoded into two triples: data triple, summary triple
- **Dictionary encoding**: Each entity is assigned a globally unique id is computed by concatenating (supernode id) \( P_s \) and a (local id) \( id_s \)
  
  Eg. \( \langle \text{Barack Obama, won, Nobel Prize} \rangle \)
  - Data triple: \( \langle 1||1, 6, 4||3 \rangle \)
  - Summary triple \( \langle 1, 6, 4 \rangle \)

Locality-aware sharding and distributed indexing of data triples
- Each data triple is hash partitioned onto atmost two slaves and indexed in six permutations (in total)
  
  Eg. triple \( \langle 1||1, 6, 4||3 \rangle \) is hashed on to slaves \( 1 \mod n \) and \( 4 \mod n \) (for \( n \)-slaves)
Indexing

RDF graph partitioning
- using a locality-based non-overlapping partitioning algorithm
- i.e. Each \(s\) (or \(o\)) of RDF triple \(\langle s, p, o \rangle\) mapped to one supernode \(P_s\) (or \(P_o\))

Triple encoding
- Each triple \(\langle s, p, o \rangle\) is encoded into two triples: data triple, summary triple
- Dictionary encoding: Each entity is assigned a globally unique id is computed by concatenating (supernode id) \(P_s\) and a (local id) \(id_s\)
  Eg. \(\langle \text{Barack Obama, won, Nobel Prize} \rangle\)
  Data triple: \(\langle 1||1, 6, 4||3 \rangle\) Summary triple \(\langle 1, 6, 4 \rangle\)

Locality-aware sharding and distributed indexing of data triples
- Each data triple is hash partitioned onto atmost two slaves and indexed in six permutations (in total)
  Eg. triple \(\langle 1||1, 6, 4||3 \rangle\) is hashed on to slaves \(1 \mod n\) and \(4 \mod n\) (for \(n\)-slaves)

Summary graph index
- Summary triples are indexed in two-permutation adjacency list for efficient graph exploration
TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing

Query Processing

In TriAD, query processing is performed in two stages

Stage 1: Summary graph query processing (join-ahead pruning)

- Performed using graph exploration at master node to generate supernode bindings

Stage 2: Data graph query processing

- Done using relational joins in a distributed setup

Inspired by the RISC style processing, we employ three physical operators:

- Distributed Index Scan (DIS): Invokes a parallel scan over a permutation list that is sharded across all slaves
- Distributed Merge Join (DMJ): If both input (or intermediate) relations are sorted according to the join key(s) in the query plan
- Distributed Hash Join (DHJ): If the input (or intermediate) relations are not sorted according to their join key(s)
In TriAD, query processing is performed in two stages

**Stage 1: Summary graph query processing (join-ahead pruning)**
- Performed using **graph exploration** at master node to generate supernode bindings

**Stage 2: Data graph query processing**
- Done using **relational joins** in a distributed setup
- Inspired by the RISC style processing, we employ three physical operators:
  - **Distributed Index Scan (DIS):** Invokes a parallel scan over a permutation list that is sharded across all slaves
  - **Distributed Merge Join (DMJ):** If both input (or intermediate) relations are sorted according to the join key(s) in the query plan
  - **Distributed Hash Join (DHJ):** If the input (or intermediate) relations are not sorted according to their join key(s)
Global Statistics & Query Optimization

Precomputed statistics

- Computed in parallel at slaves and sent to master node
- Statistics:
  - Individual cardinalities: $s, p, o$ of SPO triples
  - Pair cardinalities: $(s, o), (s, p), (p, o)$
  - Join selectivities of predicate pairs $(p_1, p_2)$
- Similar statistics are computed over summary graph

For a given query with patterns $Q \{R_1, R_2, .. R_n\}$

Stage 1: Exploration Optimization

We compute the exploration plan by using a bottom-up dynamic programming and the following cost model:

$$\text{Cost}(R_1, \ldots, R_n) \propto \text{Card}(R_1) + \sum_{i=2}^{n} \left( \text{Card}(R_i) \prod_{j=1}^{i} \text{Sel}(R_i, R_j) \right)$$  (1)
Precomputed statistics

- Computed in parallel at slaves and sent to master node
- Statistics:
  - Individual cardinalities: s, p, o of SPO triples
  - Pair cardinalities: (s, o), (s, p), (p, o)
  - Join selectivities of predicate pairs \((p_1, p_2)\)
- Similar statistics are computed over summary graph

For a given query with patterns \(Q \{R_1, R_2, ..R_n\}\)

Stage 1: Exploration Optimization

We compute the exploration plan by using a bottom-up dynamic programming and the following cost model:

\[
Cost(R_1, \ldots, R_n) \propto Card(R_1) + \sum_{i=2}^{n} \left( Card(R_i) \prod_{j=1}^{i} Sel(R_i, R_j) \right) \quad (1)
\]
TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing

Re-estimating cardinalities of a relation $R_i$

New cardinalities are re-estimated from the Stage 1 supernode bindings

- $\text{Card}(R_i) := \frac{|C^Q_s|}{|C_s|} \cdot \frac{|C^Q_o|}{|C_o|} \cdot \text{Card}(R_i)$ (on the fly computation)

where $|C_s|, |C_o|$ are the supernode bindings of $R_i$
and $|C^Q_s|, |C^Q_o|$ are the supernode bindings of $Q$ ($R_i \in Q$)

Stage 2: Global plan optimization

A global plan is computed for stage 2 using re-estimated cardinalities and the following cost model

$$\text{Cost}(Q) := \begin{cases} 
\text{Cost}(R^k_i) & \text{if } R_i \text{ denotes a DIS over permutation } k; \\
\max(\text{Cost}(Q^{\text{left}}), \text{Cost}(Q^{\text{right}})) + \text{Cost}(Q^{\text{left}} \bowtie_{op} Q^{\text{right}}) & \text{otherwise.}
\end{cases}$$

The cost model captures the multi-threaded and distributed execution framework
Re-estimating cardinalities of a relations $R_i$

New cardinalities are re-estimated from the Stage 1 supernode bindings

$\text{Card}(R_i) := \frac{|C^Q_s|}{|C_s|} \cdot \frac{|C^Q_o|}{|C_o|} \cdot \text{Card}(R_i)$ (on the fly computation)

where $|C_s|, |C_o|$ are the supernode bindings of $R_i$

and $|C^Q_s|, |C^Q_o|$ are the supernode bindings of $Q$ ($R_i \in Q$)

Stage 2: Global plan optimization

A global plan is computed for stage 2 using re-estimated cardinalities and the following cost model

$\text{Cost}(Q) := \begin{cases} 
\text{Cost}(R^k_i) & \text{if } R_i \text{ denotes a DIS over permutation } k; \\
\max(\text{Cost}(Q^{left}), \text{Cost}(Q^{right})) & \\
+\text{Cost}(Q^{left} \bowtie op Q^{right}) & \\
+\text{Cost}(Q^{left} \Rightarrow op Q^{right}) & \text{otherwise.}
\end{cases}$

The cost model captures the multi-threaded and distributed execution framework
SPARQL Query:

Find persons who are born in USA and won a prize.

SELECT ?person, ?city, ?prize WHERE {
  \(R_1\) ?person <bornIn> ?city  .
  \(R_2\) ?city <locatedIn> USA  .
  \(R_3\) ?person <won> ?prize  .
  \(R_4\) ?prize <hasName> ?name  .
}
**SPARQL Query:**
Find persons who are born in USA and won a prize.

SELECT ?person, ?city, ?prize WHERE {
  R1 ?person <bornIn> ?city .
  R2 ?city <locatedIn> USA .
  R3 ?person <won> ?prize .
  R4 ?prize <hasName> ?name .
}
**SPARQL Query:**
Find persons who are born in USA and won a prize.

```
SELECT ?person, ?city, ?prize WHERE {
  R1 ?person <bornIn> ?city .
  R2 ?city <locatedIn> USA .
  R3 ?person <won> ?prize .
  R4 ?prize <hasName> ?name .
}
```

**Summary Graph**

**Stage 1** (Optimization)

**Global Plan**

**Exploration**

**Supernode Bindings:**
- ?person : P1, P2, P4
- ?city : P1, P2, P4
- ?prize : P2, P4
Partial Results

Slave 1

Stage 2: Distributed query execution
Evaluation

We compared performance of TriAD with the following state-of-the-art systems

► Centralized systems – RDF-3X, MonetDB-RDF, BitMat
► Distributed systems – H-RDF-3X, Trinity.RDF, 4store, SHARD, Spark

Datasets:

► (Synthetic) LUBM 160 – 28 Million triples, 16GB raw data
► (Synthetic) LUBM 10240 – 1.8 Billion triples, 730GB raw data
► (Real world) BTC 2012 – 1.4 Billion triples, 231 GB raw data
► (Synthetic) WSDTS – 109 Million triples, 15 GB raw data
► Benchmark queries for LUBM, BTC, & WSDTS datasets

System Setup:

► TriAD, TriAD-SG is implemented in C++
► Cluster setup: 12-nodes, 48GB RAM, 2 quad-core CPUs of 2.4GHz (HT enabled)
TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing

Evaluation - Large datasets

LUBM 10240 Dataset – 1.8 Billion triples (Query Performance in milli seconds)

<table>
<thead>
<tr>
<th>Queries</th>
<th>Characteristics</th>
<th>TriAD</th>
<th>TriAD-SG</th>
<th>Trinity.RDF</th>
<th>H-RDF-3X</th>
<th>RDF-3X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Selective (6 joins)</td>
<td>7,631</td>
<td>2,146</td>
<td>12,648</td>
<td>2.3E6</td>
<td>1.7E5</td>
</tr>
<tr>
<td>Q2</td>
<td>Non-Selective (1 join)</td>
<td>1,663</td>
<td>2,025</td>
<td>6,018</td>
<td>5.3E5</td>
<td>4,095</td>
</tr>
<tr>
<td>Q4</td>
<td>Selective (5 joins)</td>
<td>2.1</td>
<td>1.3</td>
<td>5</td>
<td>166</td>
<td>1</td>
</tr>
<tr>
<td>Q7</td>
<td>Selective (6 joins)</td>
<td>14,895</td>
<td>16,863</td>
<td>31,214</td>
<td>2.3E6</td>
<td>2.1E5</td>
</tr>
</tbody>
</table>

Our Approach
TriAD is a fast distributed RDF engine built on top of asynchronous communication layer and multi-threaded execution framework

- efficient distributed and parallel join executions
- join-ahead pruning technique via graph summarization helps in pruning dangling triples and making query processing efficient
- distributed- and join-ahead pruning aware query optimizer
- so far reported fastest runtimes over three benchmark datasets: LUBM, BTC, WSDTS
TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing

Summary

TriAD is a fast distributed RDF engine built on top of **asynchronous communication layer** and **multi-threaded execution** framework

- efficient **distributed and parallel** join executions
- **join-ahead pruning technique via graph summarization** helps in pruning dangling triples and making query processing efficient
- **distributed- and join-ahead pruning aware** query optimizer
- so far reported fastest runtimes over three benchmark datasets: LUBM, BTC, WSDTS

 Questions & Thank You!!