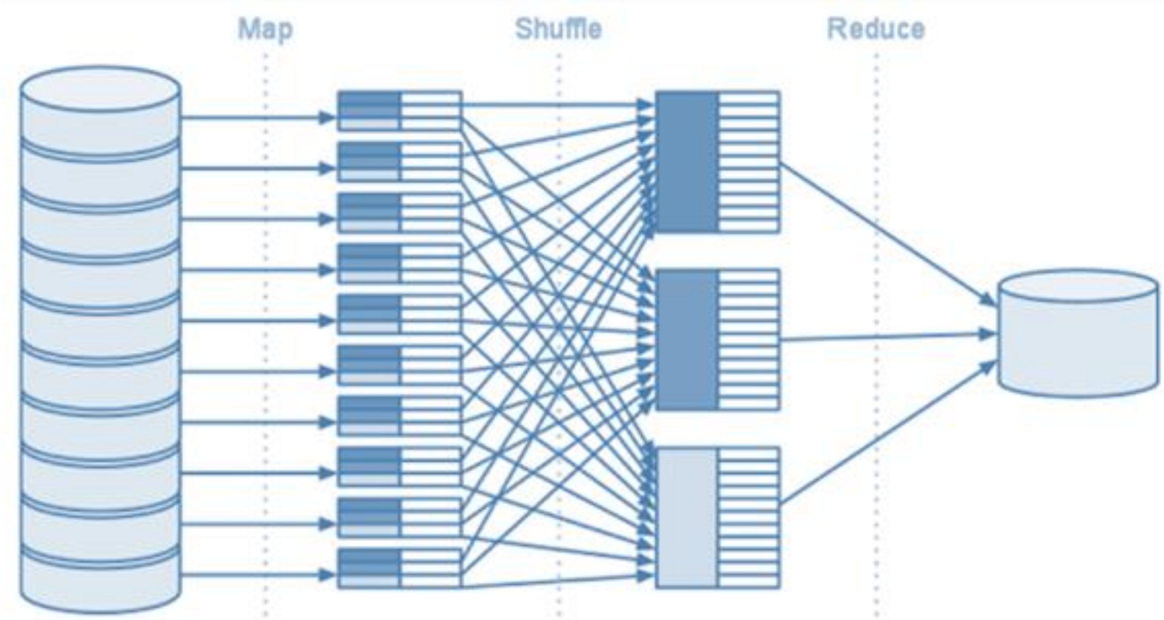
MapReduce Join Algorithms for RDF

Albert Haque
Research in Bioinformatics & Semantic Web Lab
University of Texas at Austin
October 11, 2013

Background: MapReduce

- Partition Function
  - Hashes and sorts keys; determines which reducer to send data to
- Shuffle Stage
  - Key/value pair moves from map node to reducer node



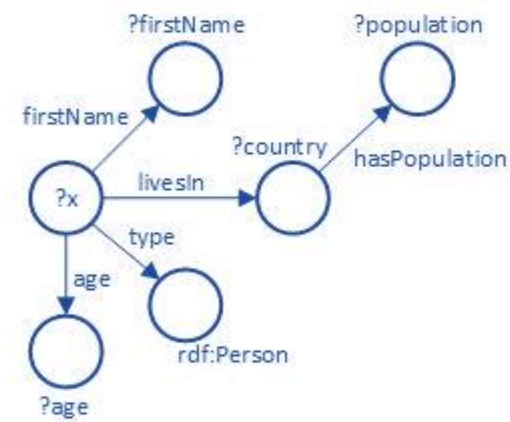
Background: Cloud Triple Store

- Property table
  - Subject as row key
  - Predicates (objects) as columns
  - Objects (predicates) as values

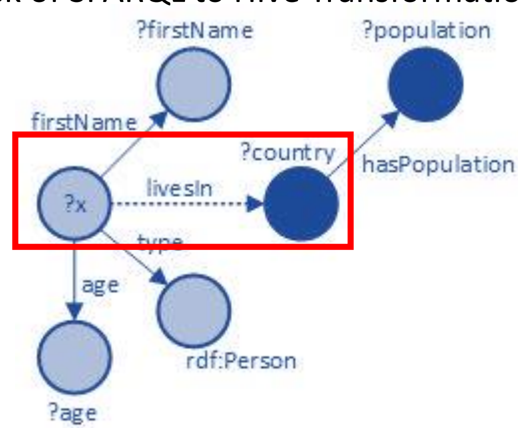
```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?firstName, ?age, ?population
WHERE {
          ?x rdfs:firstName ?firstName .
          ?x rdfs:age ?age .
          ?x rdf:type rdf:Person .
          ?x rdfs:livesIn ?country .
          ?country rdf:population ?population
}
```

- Need to join two subjects

SPARQL Query



Task of SPARQL to Hive Transformation



P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. Keppmann, D. Miranker, J. Sequeda, and M. Wylot. *NoSQL Databases for RDF: An Empirical Evaluation.* Proceedings of the 12th International Semantic Web Conference (ISWC). Oct 2013.

Algorithms

1. Map-Side Join
2. Reduce-Side Join
3. Semi-Join
4. Repartition Join

Map-Side Join

<u>Pre-Map Phase</u>

1. Each table is split into same number of partitions
2. Sort each table by the join key
3. All records for a particular key must be in same partition
   - Typically achieved by running a Reduce job beforehand

<u>Map Phase</u>

- For each partition, scan the data and join data based on key
- Emit resulting tuple

<u>Reduce Side</u>

- None

White, T. *Hadoop: The Definitive Guide, 2nd Edition*. O'Reilly Media/Yahoo Press. September 2010.

Reduce-Side Join

- Assume $R \bowtie S$
- <u>Map Phase</u>:
  - Key ***k***: Both datasets ***R*** and ***S*** have their join attributes extracted
  - Value ***v***: Rows from ***R*** or ***S*** with join attribute = ***k***
  - Tag ***t***: Identifies which dataset ***(k,v)*** belongs to
- <u>Reduce Phase</u>:

  for each $\boldsymbol{k_1}$ with ***t=R***:

      for each $\boldsymbol{k_2}$ *with* ***t=S****:*

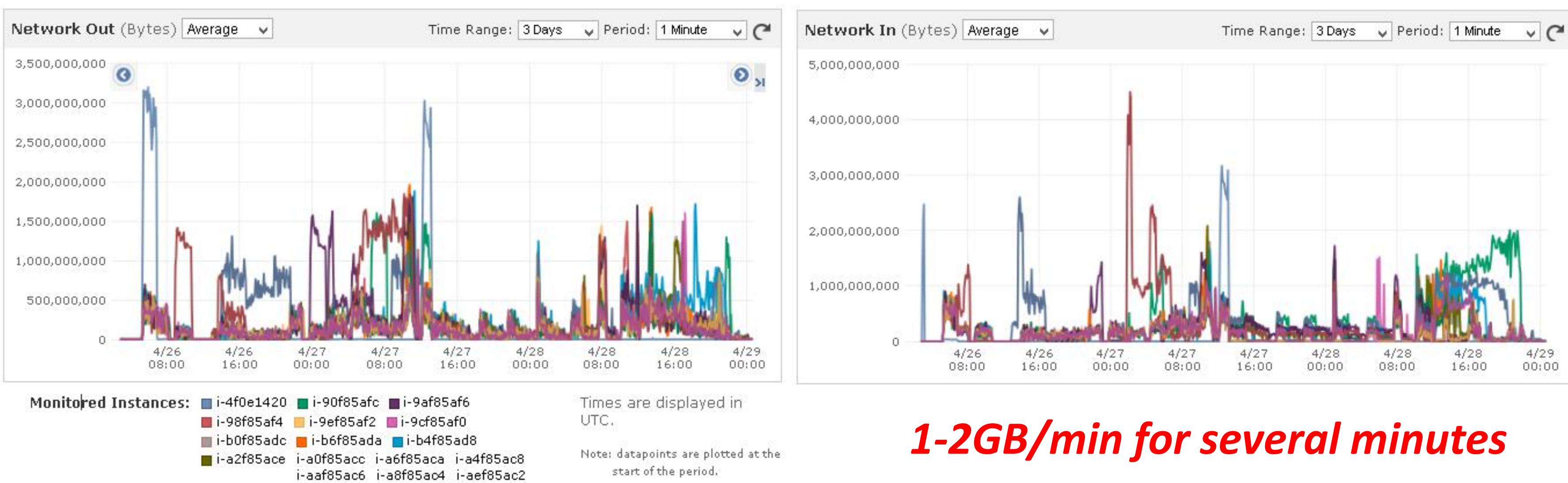
          if $\boldsymbol{k_1} = \boldsymbol{k_2}$:

              $emit(\boldsymbol{k_1}, \boldsymbol{v_1}, \boldsymbol{v_2})$

Reduce-Side Join

- Pros:
  - Easiest to implement
- Problems:
  1. Data is scattered across HDFS
  2. Sends entire dataset across network

Hadoop, HBase, & Hive – BSBM 1 billion triples, 16 nodes on AWS



1-2GB/min for several minutes

Semi-Join

- When ***R*** is large, there are many records in ***R*** that may not be referenced in ***S*** (assuming $R \bowtie S$)
- Semi-join dramatically reduces the data sent over network
- Requires 3 MapReduce jobs

  Job 1: Get a list of unique join keys, ***S.uk***. (Map+Reduce)

  Job 2: Load ***S.uk*** into memory and loop through ***R***. If a record's key is found in ***S.uk***, emit it. Now we have a list of records in ***R*** to be joined.

  Job 3: Use a broadcast join to perform the join with ***S***

Broadcast Join: If $|R| \ll |S|$ then send ***R*** to all mapper nodes

Blanas, S., et al. A comparison of Join Algorithms for Log Processing in MapReduce. SIGMOD 2010.

Semi-Join

Phase 1: Extract unique join keys in L to a single file L.uk

Map (K: null, V : a record from an L split)

join key ← extract the join column from V

if *join_key* **not in** *unique_key_table* **then**

add *join_key* to *unique_key_table*

emit (*join_key*, null)

Reduce (K': a unique join key from table L, *LIST_V'*: a list of null)

emit (K', null)

Phase 2: Use L.uk to filter referenced R records; generate a file R_i for each R split

Init ()

ref_keys ← load ***L.****uk* from phase 1 to a hash table

Map (K: null, V : a record from an R split)

join_col ← extract join column from ***V***

if join_col in ref keys **then**

emit (null, V)

Phase 3: Broadcast all R_i to each L split for the final join

Blanas, S., et al. A comparison of Join Algorithms for Log Processing in MapReduce. SIGMOD 2010.

Repartition Join

- Uses Compound Keys: **tag+key**
  - Where **tag** is an identifier for the parent table

(key,value) = (tag+key, value) = (t1albert, haque)

- Partition phase hashes the key part of the compound key
  - Guarantees tuples with same join key are sent to same reducer
- Intermediate data is sorted only by key part
- We load the smaller relation into memory by using the tag portion of compound key and perform the join

Evaluation Matrix

Data Model	Algorithm	BSBM-Q1	BSBM-Q2	…	BSBM-Q12	DBP-Q1	…	DBP-Q20	Custom Q1
SOP	Map	Timeout	Timeout		10.4 sec				40.3 sec
SOP	Reduce	5.42 sec	3.01 sec		4.42 sec	5.29 sec		6.98 sec	0.61 sec
SOP	Semi								
SOP	Repartition								
SOP	Map+Semi								
SPO	Map	Timeout	Timeout		310 sec				
SPO	Reduce								
SPO	Semi								
SPO	Repartition								
SPO	Map+Semi								

Dimensions

Selected for :

- Join Algorithm {Map, Reduce, Semi, Repartition}
- Data Model {SOP, SPO}

Additional dimensions to consider:

- Dataset {Berlin, DBpedia}
- Query $\{Q_1, Q_2, \ldots, Q_n\}$
- Cluster Size $\{1,2,4, \ldots, 2^n\}$
- Dataset Size {10 GB, 100 GB, 1 TB, 10 TB}

Research Questions

Nine Fundamental Query Join Patterns $\{S, P, O\} \times \{S, P, O\}$

- SS-Join, SP-Join, SO-Join, PS-Join, PP-Join, etc.
- How do different join algorithms perform on each of the above?

Data Models

- Are there benefits to having SPO vs. SOP?
- How do bloom filters affect performance on each schema?

- In the future, what should researchers focus their query optimizers on?
- Which join should be used for which type of queries?