Processing Joins over Big Data in MapReduce

Christos Doulkeridis

Department of Digital Systems
School of Information and Communication Technologies
University of Piraeus
http://www.ds.unipi.gr/cdoulk/
cdoulk@unipi.gr

12th Hellenic Data Management Symposium (HDMS’14), 24 July 2014
Aims and Scope

• Provide an overview of join processing in MapReduce/Hadoop, focusing on complex join types, besides equi-joins
  – e.g. theta-join, similarity join, top-k join, k-nn join …
  – “On top of” MapReduce
  – Binary joins

• Identify common techniques (at high level) that can be used as building blocks when designing a parallel join algorithm
Not Covered in this Tutorial…

- Distributed platforms for Big Data management other than Hadoop (e.g. Storm, NoSQL stores, …)
- Approaches that rely on changing the internal mechanism of Hadoop (e.g. changing data layouts, loop-aware Hadoop, etc.)
- Multi-way joins (e.g. check out the work by Afrati et al.)

- Check out related tutorials
  - MapReduce Algorithms for Big Data Analytics
    [K.Shim@VLDB12]
  - Efficient Big Data Processing in Hadoop MapReduce
    [J.Dittrich@VLDB12]
Related Research Projects

• *CloudIX*: Cloud-based Indexing and Query Processing
  – Funding program: MC-IEF, 2011-2013
  – [https://research.idi.ntnu.no/cloudix/](https://research.idi.ntnu.no/cloudix/)

• *Roadrunner*: Scalable and Efficient Analytics for Big Data
  – Open positions: Post-doc, Phd/Msc
Outline

• MapReduce basics
  – Map-side and Reduce-side join

• Join types and algorithms
  – Equi-joins
  – Theta-joins
  – Similarity joins
  – Top-k joins
  – kNN joins

• Important phases of join algorithms
  – Pre-processing, pre-filtering, partitioning, replication, load balancing

• Summary & open research directions
MapReduce Basics
# MapReduce vs. RDBMS

<table>
<thead>
<tr>
<th>Traditional RDBMS</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data size</td>
<td>Gigabytes</td>
</tr>
<tr>
<td>Access</td>
<td>Interactive and batch</td>
</tr>
<tr>
<td>Updates</td>
<td>Read and write many times</td>
</tr>
<tr>
<td>Structure</td>
<td>Static schema</td>
</tr>
<tr>
<td>Integrity</td>
<td>High</td>
</tr>
<tr>
<td>Scaling</td>
<td>Nonlinear</td>
</tr>
</tbody>
</table>

*Source: T. White “Hadoop: The Definitive Guide”*
Hadoop Distributed File System (HDFS)

Source: http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/
MapReduce

• A programming framework for parallel and scalable processing of huge datasets

• A job in MapReduce is defined as two separate phases
  – Function Map (also called mapper)
  – Function Reduce (also called reducer)

• Data is represented as key-value pairs
  – Map (k1, v1) → List (k2, v2)
  – Reduce (k2, List(v2)) → List (k3, v3)

• The data output by the Map phase are partitioned and sort-merged in order to reach the machines that compute the Reduce phase
  – This intermediate phase is called Shuffle
**Doc1**
Brazil, Germany, Argentine

**Doc2**
Chile, Germany, Brazil

**Doc3**
Greece, Japan, Brazil

**Doc4**
Chile, Germany,

**Doc5**
Chile, Brazil, Germany

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brazil</td>
<td>1</td>
</tr>
<tr>
<td>Chile</td>
<td>1,1,1</td>
</tr>
<tr>
<td>Germany</td>
<td>1,1,1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Brazil</td>
<td>1</td>
</tr>
<tr>
<td>Argentina</td>
<td>1</td>
</tr>
<tr>
<td>Japan</td>
<td>1</td>
</tr>
<tr>
<td>Germany</td>
<td>1,1,1,1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Chile</td>
<td>1</td>
</tr>
<tr>
<td>Germany</td>
<td>1</td>
</tr>
<tr>
<td>Chile</td>
<td>1</td>
</tr>
<tr>
<td>Brazil</td>
<td>1</td>
</tr>
<tr>
<td>Germany</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brazil</td>
<td>4</td>
</tr>
<tr>
<td>Chile</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Brazil</td>
<td>1</td>
</tr>
<tr>
<td>Argentina</td>
<td>1</td>
</tr>
<tr>
<td>Germany</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Implementation (High Level)

mapper (filename, file-contents):
    for each word in file-contents:
        output (word, 1)

reducer (word, values):
    sum = 0
    for each value in values:
        sum = sum + value
    output (word, sum)
MapReduce/Hadoop: 6 Basic Steps

1. Input Reader

- Reads data from files and transforms them to **key-value** pairs
  - It is possible to support different input sources, such as a database or main-memory
- The data form **splits** which is the unit of data handled by a map task
  - A typical size of a split is the size of a block (default: 64MB in HDFS, but is customizable)
2. Map Function

- Takes as input a key-value pair from the Input Reader
- Runs the code (logic) of the Map function on the pair
- Produces as result a new key-value pair

- The results of the Map function are placed in a buffer in memory, and written on disk when full (e.g. 80%) *spill files*
- The files are merged in one sorted file
3. Combiner Function

- Its use is optional
- Used when
  - The Map function produces many multiple intermediate keys
  - The Reduce function is commutative \((a + b = b + a)\) and associative \((a + b) + c = a + (b + c)\)
- In this case, the Combiner function performs partial merging so that pairs with the same key will be processed as a group by a Reduce task

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brazil</td>
<td>1</td>
</tr>
<tr>
<td>Germany</td>
<td>1</td>
</tr>
<tr>
<td>Argentine</td>
<td>1</td>
</tr>
<tr>
<td>Chile</td>
<td>1</td>
</tr>
<tr>
<td>Germany</td>
<td>1</td>
</tr>
<tr>
<td>Brazil</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brazil</td>
<td>2</td>
</tr>
<tr>
<td>Germany</td>
<td>2</td>
</tr>
<tr>
<td>Argentine</td>
<td>1</td>
</tr>
<tr>
<td>Chile</td>
<td>1</td>
</tr>
</tbody>
</table>
4. Partition Function

• **Default**: a hash function is used to partition the result of Map tasks to Reduce tasks
  – Usually, works well for load balancing

• However, it is often useful to employ other partition functions
  – Such a function can be user-defined

![Diagram of partitioning partition functions]

- Brazil
- Germany
- Argentine
- Chile
- Greece
- …

"R"
5. Reduce Function

- The Reduce function is called once for each discrete key and is applied to all values associated with the key
  - All pairs with the same key will be processed as a group
- The input to each Reduce task is given in increased key order
- It is possible for the user to define a comparator which will be used for sorting
6. Output Writer

• Responsible for writing the output to secondary storage (disk)
  – Usually, this is a file
  – However, it is possible to modify the function to store data, e.g., in a database
MapReduce Advantages

- Scalability
- Fault-tolerance
- Flexibility
- Ease of programming
- ...
10 Weaknesses/Limitations of MapReduce

1. Efficient access to data
2. High communication cost
3. Redundant processing
4. Recomputation
5. Lack of early termination
6. Lack of iteration
7. Fast computation of approximate results
8. Load balancing
9. Real-time processing
10. Lack of support for operators with multiple inputs

M-way Operations in MapReduce

• Not naturally supported
  – Other systems support such operations by design (e.g., Hyracs@UCI, Storm)

• How in MapReduce?
  – Two families of techniques (high level classification)
    • Map-side join
    • Reduce-side join
### Map-side Join

#### Table L
<table>
<thead>
<tr>
<th>Uid</th>
<th>Time</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19:45</td>
<td>Update</td>
</tr>
<tr>
<td>1</td>
<td>21:12</td>
<td>Read</td>
</tr>
<tr>
<td>2</td>
<td>21:11</td>
<td>Read</td>
</tr>
<tr>
<td>2</td>
<td>23:45</td>
<td>Write</td>
</tr>
<tr>
<td>3</td>
<td>23:09</td>
<td>Insert</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Table R
<table>
<thead>
<tr>
<th>Uid</th>
<th>Uname</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cdoulk</td>
<td>Lecturer</td>
</tr>
<tr>
<td>2</td>
<td>jimper0</td>
<td>Postgrad</td>
</tr>
<tr>
<td>3</td>
<td>spirosoik</td>
<td>Postgrad</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Diagram

- **L** → **R**
  - **M**
    - **cdoulk** → **Update**
    - **cdoulk** → **Read**
    - **Jimper0** → **Read**
    - **Jimper0** → **Write**
    - **Spirosoik** → **Insert**
Map-side Join

- Operates on the map side, no reduce phase
- Requirements for each input
  - Divided into the same number of partitions
  - Sorted by the join key
  - Has the same number of keys

- Advantages
  - Very efficient (no intermediate data, no shuffling, simply scan and join)

- Disadvantages
  - Very strict requirements, in practice an extra MR job is necessary to prepare the inputs
  - High required memory (buffers both input partitions)

Source: T.White “Hadoop: The Definitive Guide”
Reduce-side Join

<table>
<thead>
<tr>
<th>Uid</th>
<th>Time</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19:45</td>
<td>Update</td>
</tr>
<tr>
<td>2</td>
<td>21:11</td>
<td>Read</td>
</tr>
<tr>
<td>1</td>
<td>21:12</td>
<td>Read</td>
</tr>
<tr>
<td>3</td>
<td>23:09</td>
<td>Insert</td>
</tr>
<tr>
<td>2</td>
<td>23:45</td>
<td>Write</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uid</th>
<th>Uname</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cdoulk</td>
<td>Lecturer</td>
</tr>
<tr>
<td>2</td>
<td>jimper0</td>
<td>Postgrad</td>
</tr>
<tr>
<td>3</td>
<td>spirosoik</td>
<td>Postgrad</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\( L \bowtie_{uid=uid} R \)

- **L**: Left side
- **R**: Right side
- **M**: Merge node
- **L** and **R** nodes connect through **M** nodes representing actions.
- **Unname** and **Action** for each action.
- **Uid**, **Time**, **Action** for each user action.
Reduce-side Join

• a.k.a. repartition join
• The join computation is performed on the reduce side

• Advantages
  – The most general method (similar to a traditional parallel join)
  – Smaller memory footprint (only tuples of one dataset with the same key)

• Disadvantages
  – Cost of shuffling, I/O and communication costs for transferring data to reducers
  – High memory requirements for skewed data

Source: T.White “Hadoop: The Definitive Guide”
Join Types and Algorithms
Example: Equi-Join

<table>
<thead>
<tr>
<th>Uid</th>
<th>Time</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19:45</td>
<td>Update</td>
</tr>
<tr>
<td>2</td>
<td>21:11</td>
<td>Read</td>
</tr>
<tr>
<td>1</td>
<td>21:12</td>
<td>Read</td>
</tr>
<tr>
<td>3</td>
<td>23:09</td>
<td>Insert</td>
</tr>
<tr>
<td>2</td>
<td>23:45</td>
<td>Write</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uid</th>
<th>Uname</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cdoulk</td>
<td>Lecturer</td>
</tr>
<tr>
<td>2</td>
<td>jimper0</td>
<td>Postgrad</td>
</tr>
<tr>
<td>3</td>
<td>spirosoik</td>
<td>Postgrad</td>
</tr>
</tbody>
</table>

$L \bowtie_{uid=uid} R$
Equi-Joins: Broadcast Join

- Map-only job
- Assumes that \(|R| \ll |L|\)

Blanas et al. SIGMOD’10
Equi-Joins: Broadcast Join

• Advantages
  – Efficient: only map-phase
    • Load the small dataset $R$ in hash table: faster probes
  – No pre-processing (e.g., sorting)

• Disadvantages
  – One dataset must be quite small
    • Fit in memory
    • Distribute to all mappers
Distinct keys

Equi-Joins: Semi-join

R' contains only those keys in L.uid

Broadcast Join

Blanas et al. SIGMOD’10
Equi-Joins: Semi-join

• Advantages
  – When $R$ is large, reduces size of intermediate data and communication costs
  – Useful when many tuples of one dataset do not join with tuples of the other dataset

• Disadvantages
  – Three MR jobs $\rightarrow$ overheads: job initialization, communication, local and remote I/O
  – Multiple accesses of datasets
Equi-Joins: Per-Split Semi-join

Blanas et al. SIGMOD’10
Equi-Joins: Per-Split Semi-join

• Advantages (compared to Semi-join)
  – Makes the 3rd phase cheaper, as it moves only the records in \( R \) that will join with each split of \( L \)

• Disadvantages (compared to Semi-join)
  – First two phases are more complicated
Other Approaches for Equi-Joins

- Map-Reduce-Merge [Yang et al. SIGMOD’07]
- Map-Join-Reduce [Jiang et al. TKDE’11]
- Llama [Lin et al. SIGMOD’11]
- Multi-way Equi-Join [Afrati et al. TKDE’11]
Example: Theta-Join

<table>
<thead>
<tr>
<th>Uid</th>
<th>Time</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19:45</td>
<td>Update</td>
</tr>
<tr>
<td>2</td>
<td>21:11</td>
<td>Read</td>
</tr>
<tr>
<td>1</td>
<td>21:12</td>
<td>Read</td>
</tr>
<tr>
<td>3</td>
<td>23:09</td>
<td>Insert</td>
</tr>
<tr>
<td>2</td>
<td>23:45</td>
<td>Write</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uid</th>
<th>Uname</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cdoulk</td>
<td>Lecturer</td>
</tr>
<tr>
<td>2</td>
<td>jimper0</td>
<td>Postgrad</td>
</tr>
<tr>
<td>3</td>
<td>spirosikoik</td>
<td>Postgrad</td>
</tr>
</tbody>
</table>

\[ S \bowtie S.uid \leq T.uid \]

**Challenge:** How to map the inequality join condition to a key-equality based computer paradigm
Join Matrix Representation

- $M(i,j)=\text{true}$, if the $i$-th tuple of $S$ and the $j$-th tuple of $T$ satisfy the join condition
- **Problem**: find a mapping of join matrix cells to reducers that minimizes job completion time
  - Each join output tuple should be produced by exactly one reducer

Okcan et al. SIGMOD’11
Matrix-to-Reducer Mapping

<table>
<thead>
<tr>
<th>Standard</th>
<th>Random</th>
<th>Balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max-reducer-input=5</td>
<td>Max-reducer-input=8</td>
<td>Max-reducer-input=5</td>
</tr>
<tr>
<td>Max-reducer-output=6</td>
<td>Max-reducer-output=4</td>
<td>Max-reducer-output=4</td>
</tr>
</tbody>
</table>

Okcan et al. SIGMOD’11
MapReduce Algorithm: 1-Bucket-Theta

- Given a matrix-to-reducer mapping
- For each incoming S-tuple, say $x \in S$
  - Assign a random matrix row from $1 \ldots |S|$
  - For all regions that correspond to this row
    - Output $(\text{regionID}, (x, \text{“S”}))$
- For each incoming T-tuple, say $y \in T$
  - Assign a random matrix column from $1 \ldots |T|$
  - For all regions that correspond to this column
    - Output $(\text{regionID}, (y, \text{“T”}))$

---

Okcan et al. SIGMOD’11
In the paper...

- How to perform near-optimal partitioning with strong optimality guarantees
  - For cross-product $S \times T$
- More efficient algorithms (M-Bucket-I, M-Bucket-O) when more statistics are available
Other Approaches for Theta-Joins

- Multi-way Theta-joins [Zhang et al. VLDB’12]
- Binary Theta-joins [Koumarelas et al. EDBT’14]
Example: (Set) Similarity Join

\[
\begin{array}{|c|c|c|}
\hline
\text{Rid} & \text{a} & \text{b} \\
\hline
1 & A B C & \ldots \\
2 & D E F & \ldots \\
3 & C F & \ldots \\
4 & E C D & \ldots \\
5 & B A F & \ldots \\
\ldots & \ldots & \ldots \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{Rid} & \text{a} & \text{c} \\
\hline
1 & G H I & \ldots \\
2 & A F G & \ldots \\
3 & B D F & \ldots \\
\ldots & \ldots & \ldots \\
\hline
\end{array}
\]

Example (strings):
S1: “I will call back” → [I, will, call, back]
S2: “I will call you soon” → [I, will, call, you, soon]
e.g. \( \text{sim}() \): Jaccard, \( \tau = 0.4 \)
\( \text{sim}(S1,S2) = \frac{3}{6} = 0.5 > 0.4 \)
Background: Set Similarity Joins

• Rely on filters for efficiency, such as *prefix filtering*
  – *Global token ordering*, sort tokens based on increasing frequency order
  – $\text{Jaccard}(s_1, s_2) \geq \tau \iff \text{Overlap}(s_1, s_2) \geq a$
    • $a = \tau / (1 + \tau) \times (|s_1| + |s_2|)$
  – *Similar records need to share at least one common token in their prefixes*

• Example:
  – $s_1 \{A, B, C, D, E\}$
  – Prefix length of $s_1$: $|s_1| - \text{ceil}(\tau|s_1|) + 1$
  – For $\tau=0.9$: $5 - \text{ceil}(0.9 \times 5) + 1 = 1 \rightarrow$ Prefix of $s_1$ is $\{A\}$
  – For $\tau=0.8$: $5 - \text{ceil}(0.8 \times 5) + 1 = 2 \rightarrow$ Prefix of $s_1$ is $\{A, B\}$
  – For $\tau=0.6$: $5 - \text{ceil}(0.6 \times 5) + 1 = 3 \rightarrow$ Prefix of $s_1$ is $\{A, B, C\}$
Set Similarity Joins

• Solves the problem of Set Similarity Self-Join in 3 stages
  – Can be implemented as 3 (or more) MR jobs
    • #1: Token Ordering
    • #2: RID-pair Generation
    • #3: Record Join
Set Similarity Joins
Stage 1: Token Ordering

Vernica et al. SIGMOD’10
Set Similarity Joins
Stage 2: RID-pair Generation

Vernica et al. SIGMOD’10
Set Similarity Joins
Stage 3: Record Join
Other Approaches for Similarity Joins

- V-SMART-Join [Metwally et al. VLDB’12]
- SSJ-2R [Baraglia et al. ICDM’10]
- Silva et al. [Cloud-I’12]
- Top-k similarity join [Kim et al. ICDE’12]
- Fuzzy join [Afrati et al. ICDE’12]
Example: Top-k Join

<table>
<thead>
<tr>
<th>Supplier</th>
<th>Product</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Monitor</td>
<td>100</td>
</tr>
<tr>
<td>S2</td>
<td>CPU</td>
<td>160</td>
</tr>
<tr>
<td>S3</td>
<td>CPU</td>
<td>120</td>
</tr>
<tr>
<td>S4</td>
<td>HDD</td>
<td>190</td>
</tr>
<tr>
<td>S5</td>
<td>HDD</td>
<td>90</td>
</tr>
<tr>
<td>S6</td>
<td>DVD-RW</td>
<td>200</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer</th>
<th>Product</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Monitor</td>
<td>800</td>
</tr>
<tr>
<td>C2</td>
<td>CPU</td>
<td>240</td>
</tr>
<tr>
<td>C3</td>
<td>HDD</td>
<td>500</td>
</tr>
<tr>
<td>C4</td>
<td>CPU</td>
<td>550</td>
</tr>
<tr>
<td>C5</td>
<td>HDD</td>
<td>700</td>
</tr>
<tr>
<td>C6</td>
<td>Monitor</td>
<td>600</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\[ \text{Topk}(L \bowtie_{L\text{.product}=R\text{.product}} R) \]
RanKloud

• Basic ideas
  – Generate statistics at runtime
    (adaptive sampling)
  – Calculate threshold that
    identifies candidate tuples for
    inclusion in the top-k set
  – Repartition data to achieve load
    balancing

• Cannot guarantee
  completeness
  – May produce fewer than k
    results

• Probably requires 2
  MapReduce jobs

Candan et al. *IEEE Multimedia 2011*
Top-k Joins: Our Approach

• Basic features
  – Exploit histograms built seamlessly during data upload to HDFS
  – Compute bound
  – Access only part of the input data (Map phase)
  – Perform the top-k join in a load-balanced way (Reduce phase)
• …working paper
Other Approaches for Top-k Joins

• BFHM Rank-Join in HBase [Ntarmos et al. VLDB’14]
  – Tomorrow, HDMS session 5, 10:50-12:30 😊
Example: kNN-Join

\[
\text{knnJ}(R,S) = \{(r, \text{knn}(r,S)) \text{ for all } r \in R\}
\]

<table>
<thead>
<tr>
<th>Sid</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>43</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>45</td>
<td>76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rid</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>45</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
kNN Joins in MapReduce

• Straightforward solution: *Block nested loop join (H-BNLJ)*
  - 1\textsuperscript{st} MapReduce job
    • Partition R and S in n equi-sized blocks
    • Every pair of blocks is partitioned into a bucket (n\(^2\) buckets)
    • Invoke r=n\(^2\) reducers to do kNN join (nk candidates for each record)
  - 2\textsuperscript{nd} MapReduce job
    • Partition output records of 1\textsuperscript{st} phase by record ID
    • In the reduce phase, sort in ascending order of distance
    • Then, report top-k results for each record

• Improvement (*H-BRJ*):
  - Build an index for the local S block in the reducer, to find kNN efficiently

---

Zhang et al. *EDBT’12*
H-zkNNJ

• Exact solution is too costly! go for approximate solution instead

• Solves the problem in 3 phases
  – #1: Construction of shifted copies of $R$ and $S$
  – #2: Compute candidate k-NN for each record $r \in R$
  – #3: Determine actual k-nn for each $r$

Zhang et al. EDBT'12
H-zkNNJ: Phase 1

Stored locally

Estimated quantiles $A[0]...A[n]$ for partitioning $R_i, S_i$
H-zkNNJ: Phase 2

retrieve $C_i(r)$ for all $r \in R_{i,j}$, $j \in [1, n]$

**Map**

- Partition by lines 14-15 of Alg. 2
- Partition by lines 16-19 of Alg. 2

** Reduce**

- $\alpha^n$ reducers

Zhang et al. *EDBT’12*
H-zkNNJ: Phase 3

• Output of Phase 2:
  – For each record \( r \in R \):
    • \( C(r) \) candidates

• Compute \( \text{knn}(r, C(r)) \)...trivial
Other Approaches for kNN Joins

• [Lu et al. VLDB’12]
  – Partitioning based on Voronoi diagrams
Important Phases of Join Algorithms

“How to boost the speed of my join algorithm?”
Pre-processing

• Includes techniques such as
  – Ordering [Vernica et al. SIGMOD’10]
    • Global token ordering
  – Sampling [Kim et al. ICDE’12]
    • Upper bound on the distance of k-th closest pair
  – Frequency computation [Mettawally et al. VLDB’12]
    • Partial results and cardinalities of items
  – Columnar storage, sorted [Lin et al. SIGMOD’11]
    • Enables map-side join processing
Pre-filtering

• Reduces the candidate input tuples to be processed
  – *Essential pairs* [Kim et al. ICDE’12]
    • Partitioning only those pairs necessary for producing top-k most similar pairs
Partitioning

- More advanced partitioning schemes than hashing (application-specific)
  - Theta-joins [Okcan et al. SIGMOD’11]
    - Map the join matrix to the reduce tasks
  - kNN joins [Lu et al. VLDB’12]
    - Z-ordering for 1-dimensional mapping
    - Range partitioning
  - Top-k joins
    - Partitioning schemes tailored to top-k queries
      - Utility-aware partitioning [RanKloud]
      - Angle-based partitioning [Doulkeridis et al. Cloudl’12]
Replication/Duplication

• In certain parallel join algorithms, replication of records is unavoidable
  – How can we control the amount of replication?

• Set-similarity join [Vernica et al. SIGMOD’10]
  – Grouped tokens

• Multi-way theta joins [Zhang et al. VLDB’12]
  – Minimize duplication of records to partitions
  – Use a multidimensional space partitioning method to assign partitions of the join space to reducers
Load Balancing

- Data skew causes unbalanced allocation of work to reducers (lack of a built-in data-aware load balancing mechanism)
  - Frequency-based assignment of keys to reducers [Vernica et al. SIGMOD’10]
  - Cardinality-based assignment [Metwally et al. VLDB’12]
  - Minimize job completion time [Okcan et al. SIGMOD’11]
  - Approximate quantiles [Lu et al. VLDB’12]
    - Partition the input relations to equi-sized partitions
## Analysis of Join Processing

<table>
<thead>
<tr>
<th>Join type</th>
<th>Approach</th>
<th>Pre-processing</th>
<th>Pre-filtering</th>
<th>Partitioning</th>
<th>Replication</th>
<th>Load balancing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equi-join</td>
<td>Map-Reduce-Merge [29]</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Map-Join-Reduce [58]</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Afrati et al. [5,6]</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>Hash-based</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Repartition join [18]</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>Hash-based</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Broadcast join [18]</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>Broadcast</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Semi-join [18]</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>Broadcast</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Per-split semi-join [18]</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>Vertical groups</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Llama [74]</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>Cover join matrix</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bounds</td>
</tr>
<tr>
<td>Theta-join</td>
<td>1-Bucket-Theta [84]</td>
<td>No</td>
<td>No</td>
<td>Cover join matrix</td>
<td>Yes</td>
<td>Bounds</td>
</tr>
<tr>
<td></td>
<td>M-Bucket [84]</td>
<td>Yes</td>
<td>No</td>
<td>Cover join matrix</td>
<td>Yes</td>
<td>Bounds</td>
</tr>
<tr>
<td></td>
<td>Zhang et al. [110]</td>
<td>No</td>
<td>No</td>
<td>Hilbert space-filling curve</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Similarity join</td>
<td>Set-similarity join [102]</td>
<td>Global token ordering</td>
<td>No</td>
<td>Hash-based</td>
<td>Grouping</td>
<td>Frequency-based</td>
</tr>
<tr>
<td></td>
<td>V-SMART-Join [81]</td>
<td>Compute frequencies</td>
<td>No</td>
<td>Hash-based</td>
<td>Yes</td>
<td>Cardinality-based</td>
</tr>
<tr>
<td></td>
<td>SSJ-2R [16]</td>
<td>Similar to [102]</td>
<td>No</td>
<td>Remainder file</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Silva et al. [97]</td>
<td>No</td>
<td>No</td>
<td>Iterative</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Top-k similarity join [61]</td>
<td>No</td>
<td>No</td>
<td>Bucket-based</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Fuzzy join [4]</td>
<td>No</td>
<td>No</td>
<td>Essential pairs</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Similarity join</td>
<td></td>
<td></td>
<td></td>
<td>No</td>
<td>Z-value based</td>
<td>Shifted copies of R, S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Quantile-based</td>
</tr>
<tr>
<td>k-NN join</td>
<td>H-zkNNJ [77]</td>
<td>No</td>
<td>No</td>
<td>Z-value based</td>
<td>Shifted copies of R, S</td>
<td>Yes</td>
</tr>
<tr>
<td>Top-k join</td>
<td>RanKloud [24]</td>
<td>Sampling</td>
<td>No</td>
<td>Utility-aware</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Doulkeridis et al. [38]</td>
<td>Sorting</td>
<td>No</td>
<td>Angle-based</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Source:** C.Doulkeridis and K.Nørvåg. A Survey of Large-Scale Analytical Query Processing in MapReduce. In *the VLDB Journal*, June 2014.
Summary and Open Research Directions
Summary

• Processing joins over Big Data in MapReduce is a challenging task
  – Especially for complex join types, large-scale data, skewed datasets, …
  – However, focus on the techniques(!), not the platform

• New techniques are required
• Old techniques (parallel joins) need to be revisited to accommodate the needs of new platforms (such as MapReduce)
• Window of opportunity to conduct research in an interesting topic
Trends

• Scalable *and efficient* platforms
  – MapReduce improvements, new platforms (lots of research projects worldwide), etc.

• **Declarative querying** instead of programming language
  – Hive, Pig (over MapReduce), Trident (over Storm)

• **Real-time and interactive** processing

• **Data visualization**

• **Main-memory databases**
References

• [Blanas et al. SIGMOD’10] A comparison of join algorithms for log processing in MapReduce.
• [Yang et al. SIGMOD’07] Map-reduce-merge: simplified relational data processing on large clusters.
• [Jiang et al. TKDE’11] MAP-JOIN-REDUCE: Toward scalable and efficient data analysis on large clusters.
• [Lin et al. SIGMOD’11] Llama: leveraging columnar storage for scalable join processing in the MapReduce framework.
• [Okcan et al. SIGMOD’11] Processing Theta-Joins using MapReduce.
• [Zhang et al. VLDB’12] Efficient Multi-way Theta-Join Processing using MapReduce.
• [Ntarmos et al. VLDB’14] Rank Join Queries in NoSQL Databases
• [Zhang et al. EDBT’12] Efficient Parallel kNN Joins for Large Data in MapReduce
• [Lu et al. VLDB’12] Efficient Processing of k Nearest Neighbor Joins using MapReduce
Thank you for your attention

More info: http://www.ds.unipi.gr/cdoulk/
Contact: cdoulk@unipi.gr

https://research.idi.ntnu.no/cloudix/
http://www.ds.unipi.gr/projects/roadrunner/