Relational Data Processing on MapReduce

Vassilis Christophides
christop@csd.uoc.gr
http://www.csd.uoc.gr/~hy562
University of Crete, Fall 2019

Peta-scale Data Analysis

12+ TBs of tweet data every day

30 billion RFID tags today (1.3B in 2005)

4.6 billion camera phones world wide

25+ TBs of log data every day generated by a new user being added every sec. for 3 years

100s of millions of GPS enabled devices sold annually

4 billion views/day

76 million smart meters in 2009... 200M by 2014

2+ billion people on the Web by end 2011
Big Data Analysis

- A lot of these datasets have some structure
  - Query logs
  - Point-of-sale records
  - User data (e.g., demographics)
  - ...

- How do we perform data analysis at scale?
  - Relational databases and SQL
  - MapReduce (Hadoop)

Relational Databases vs. MapReduce

- Relational databases:
  - Multipurpose: analysis and transactions; batch and interactive
  - Data integrity via ACID transactions
  - Lots of tools in software ecosystem (for ingesting, reporting, etc.)
  - Supports SQL (and SQL integration, e.g., JDBC)
  - Automatic SQL query optimization

- MapReduce (Hadoop):
  - Designed for large clusters, fault tolerant
  - Data is accessed in “native format”
  - Supports many query languages
  - Programmers retain control over performance
Parallel Relational Databases vs. MapReduce

- Parallel relational databases
  - Schema on “write”
  - Failures are relatively infrequent
  - “Possessive” of data
  - Mostly proprietary

- MapReduce
  - Schema on “read”
  - Failures are relatively common
  - In situ data processing
  - Open source
MapReduce: A Major Step Backwards?

- MapReduce is a step backward in database access
  - Separation of the schema from the application is good
  - Sharing across multiple MR programs is difficult
  - Declarative access languages are good
    - Does not require highly-skilled programmers
- MapReduce is a poor implementation
  - Brute force and only brute force
    - No indexes: Wasteful access to unnecessary data
  - Don’t need 1000 nodes to process petabytes
    - Parallel DBs do it in fewer than 100 nodes
- MapReduce is missing features
  - Bulk loader, indexing, updates, transactions...
  - No support for JOINs:
    - Requires multiple MR phases for the analysis

Map Reduce vs Parallel DBMS

<table>
<thead>
<tr>
<th></th>
<th>Parallel DBMS</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema Support</td>
<td>✓</td>
<td>Not out of the box</td>
</tr>
<tr>
<td>Indexing</td>
<td>✓</td>
<td>Not out of the box</td>
</tr>
<tr>
<td>Programming Model</td>
<td>Declarative (SQL)</td>
<td>Imperative (C/C++, Java, …) Extensions through Pig and Hive</td>
</tr>
<tr>
<td>Optimizations (Compression, Query Optimization)</td>
<td>✓</td>
<td>Not out of the box</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Not out of the box</td>
<td>✓</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Coarse grained techniques</td>
<td>✓</td>
</tr>
</tbody>
</table>
Database Workloads

- **OLTP (online transaction processing)**
  - Typical applications: e-commerce, banking, airline reservations
  - User facing: *real-time, low latency, highly-concurrent*
  - Tasks: relatively small set of "standard" transactional queries
  - Data access pattern: *random reads, updates, writes* (involving relatively small amounts of data)

- **OLAP (online analytical processing)**
  - Typical applications: business intelligence, data mining
  - Back-end processing: *batch workloads, less concurrency*
  - Tasks: complex *analytical queries*, often ad hoc
  - Data access pattern: *table scans*, large amounts of data involved per query

One Database or Two?

- **Downsides of co-existing OLTP and OLAP workloads**
  - Poor memory management
  - Conflicting data access patterns
  - Variable latency

- **Solution**: separate databases
  - User-facing *OLTP database* for high-volume transactions
  - Data warehouse for *OLAP workloads*
  - How do we connect the two?
OLTP/OLAP Integration

- OLTP database for user-facing transactions
  - Retain records of all activity
  - Periodic ETL (e.g., nightly)
- Extract-Transform-Load (ETL)
  - Extract records from source
  - Transform: clean data, check integrity, aggregate, etc.
  - Load into OLAP database
- OLAP database for data warehousing
  - Business intelligence: reporting, ad hoc queries, data mining, etc.
  - Feedback to improve OLTP services

OLTP/OLAP Architecture: Hadoop?

What about here?

- OLTP
- OLAP

Hadoop here?
OLTP/OLAP/Hadoop Architecture

- Why does this make sense?

ETL Bottleneck

- Reporting is often a nightly task:
  - ETL is often slow (see next picture)!
    - What happens if processing 24 h of data takes longer than 24 h?
  - Often, with noisy datasets, ETL is the analysis!
    - ETL necessarily involves brute force data scans: L, then E and T?
  - Hadoop is perfect:
    - Most likely, you already have some data warehousing solution
    - Ingest is limited by speed of HDFS
    - Scales out with more nodes
    - Massively parallel and much cheaper than parallel databases
    - Ability to use any processing tool
    - ETL is a batch process anyway!
A Closer Look at ETL

MapReduce Algorithms for Processing Relational Data
**Secondary Sorting**

- MapReduce sorts input to reducers by key
  - Values are arbitrarily ordered
- What if want to sort value also?
  - E.g., $k \rightarrow (v_1, R), (v_3, R), (v_4, R), (v_8, R)\ldots$
- Solution 1:
  - Buffer values in memory, then sort
  - Why is this a bad idea?
- Solution 2:
  - "Value-to-key conversion": extends the key with part of the value
  - Let execution framework do the sorting
  - Preserve state across multiple key-value pairs to handle processing
  - Anything else we need to do?

---

**Value-to-Key Conversion**

**Before**

$k \rightarrow (v_1, R), (v_4, R), (v_8, R), (v_3, R)\ldots$

*Values arrive in arbitrary order…*

**After**

$(k, v_1) \rightarrow (v_1, R)$  *Values arrive in sorted order…*

$(k, v_3) \rightarrow (v_3, R)$  *Process by preserving state across multiple keys!*

$(k, v_4) \rightarrow (v_4, R)$

$(k, v_8) \rightarrow (v_8, R)$

...  

- Default comparator, group comparator, and Partitioner has to be tuned to use the appropriate part of the key
Working Scenario

- Two tables:
  - User demographics (gender, age, income, etc.)
  - User page visits (URL, time spent, etc.)

- Analyses we might want to perform:
  - Statistics on demographic characteristics
  - Statistics on page visits
  - Statistics on page visits by URL
  - Statistics on page visits by demographic characteristic
  - ...

Relational Algebra

![Diagram of Relational Algebra]

- Set operations
  - Union
  - Intersection
  - Difference
  - Cartesian product

- Relational database specific operations
  - Selection
  - Projection
  - Join
  - Set division

- Set functions
  - Sum
  - Avg
  - Count
  - Any
  - Max
  - Min

[Link: www.mathcs.emory.edu/~cheung/Courses/377/Syllabus/4-RelAlg/intro.html]
Projection

Projection in MapReduce

- Easy!
  - Map over tuples, emit new tuples with the projected attributes
    - For each tuple t in R, construct a tuple t’ by eliminating those components whose attributes are not in S, emit a key/value pair (t’, t’)
  - No reducers (reducers are the identity function), unless for regrouping or resorting tuples
    - the Reduce operation performs duplicate elimination
  - Alternatively: perform in reducer, after some other processing

- Basically limited by HDFS streaming speeds
  - Speed of encoding/decoding tuples becomes important
  - Relational databases take advantage of compression
  - Semi-structured data? No problem!
Selection

\[ \sigma_C(R) \]

Selection in MapReduce

- Easy!
  - Map over tuples, emit only tuples that meet selection criteria
    - For each tuple \( t \) in \( R \), check if \( t \) satisfies \( C \) and if so, emit a key/value pair \( (t, t) \)
  - No reducers (reducers are the identity function), unless for regrouping or resorting tuples
  - Alternatively: perform in reducer, after some other processing

- Basically limited by HDFS streaming speeds:
  - Speed of encoding/decoding tuples becomes important
  - Relational databases take advantage of compression
  - Semistructured data? No problem!
Set Operations in Map Reduce

- \( R(X, Y) \cup S(Y, Z) \)
  - **Map**: for each tuple \( t \) either in \( R \) or in \( S \), emit \( (t, t) \)
  - **Reduce**: either receive \( (t, [t, t]) \) or \( (t, [t]) \)
    - Always emit \( (t, t) \)
    - We perform duplicate elimination

- \( R(X, Y) \cap S(Y, Z) \)
  - **Map**: for each tuple \( t \) either in \( R \) or in \( S \), emit \( (t, t) \)
  - **Reduce**: either receive \( (t, [t, t]) \) or \( (t, [t]) \)
    - Emit \( (t, t) \) in the former case and nothing \( (t, \text{NULL}) \) in the latter

- \( R(X, Y) - S(Y, Z) \)
  - **Map**: for each tuple \( t \) either in \( R \) or in \( S \), emit \( (t, \text{R or S}) \)
  - **Reduce**: receive \( (t, [\text{R}]) \) or \( (t, [\text{S}]) \) or \( (t, [\text{R, S}]) \)
    - Emit \( (t, t) \) only when received \( (t, [\text{R}]) \) otherwise nothing \( (t, \text{NULL}) \)

Group by… Aggregation

- Example: What is the average time spent per URL?

- In SQL:
  - **SELECT** url, AVG(time) FROM visits GROUP BY url

- In MapReduce: Let \( R(A, B, C) \) be a relation to which we apply \( Y_{A, \theta(B)}(R) \)
  - The map operation prepares the grouping (e.g., emit time, keyed by url)
  - The grouping is done by the framework
  - The reducer computes the aggregation (e.g. average)
  - Eventually, optimize with combiners
  - Simplifying assumptions: one grouping attribute and one aggregation function
Relational Joins

Types of Relationships

Many-to-Many  One-to-Many  One-to-One
Join Algorithms in MapReduce

- “Join” usually just means equi-join, but we also want to support other join predicates

- Hadoop has some built-in join support, but our goal is to understand important algorithm design principles

- Algorithms
  - Reduce-side join
  - Map-side join
  - In-memory join
    - Striped variant
    - Memcached variant

Re-Partition Join

- Each mapper processes one block (split)
- Each mapper produces the join key and the record pairs
- HDFS stores data blocks (Replicas are not shown)
- Reducers perform the actual join
- Shuffling and sorting over the network
Reduce-side Join

- Basic idea: group by join key
  - Execution framework brings together tuples sharing the same key
  - Similar to a “sort-merge join” in the database terminology

- A map function
  - Receives a record in R and S
  - Emits its join attribute value as a key and the record as a value

- A reduce function
  - Receives each join attribute value with its records from R and S
  - Perform actual join between the records in R and S

- Two variants
  - 1-to-1 joins
  - 1-to-many and many-to-many joins

Reduce-side Join: 1-to-1

Map

Reduce

Note: no guarantee if R is going to come first or S!
Reduce-side Join: 1-to-Many

Map

Reduce

What's the problem?

R is the one side, S is the many

Reduce-side Join: Value-to-Key Conversion

In reducer...

Buffer all values in memory, pick out the tuple from R, and then cross it with every tuple from S to perform the join

New key encountered: hold in memory

Cross with records from other set

New key encountered: hold in memory

Cross with records from other set
### Reduce-side Join: Many-to-Many

**In reducer…**

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td></td>
</tr>
<tr>
<td>S9</td>
<td></td>
</tr>
</tbody>
</table>

- What’s the problem?
  - R is the smaller dataset

![Diagram](image)

- Hold in memory
- Cross with records from other set

### Map-side Join: Basic Idea

- What are the limitations of reduce-side joins?
  - Both relations are transferred over the network

- Assume two datasets are sorted by the join key:

![Diagram](image)

- A sequential scan through both relations to join: called a "sort-merge join" in database terminology
Map-side Join: Parallel Scans

- If datasets are sorted by join key, join can be accomplished by a scan over both relations

```
<table>
<thead>
<tr>
<th>EMP</th>
<th>DEPTNO</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>GALLOWAY</td>
<td>10</td>
<td>TECHWRITER</td>
</tr>
<tr>
<td>DULLEY</td>
<td>20</td>
<td>ADMIN</td>
</tr>
<tr>
<td>REITER</td>
<td>30</td>
<td>HR</td>
</tr>
<tr>
<td>TAYLOR</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>HATHAWAY</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>PREVATT</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>DEPT</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>TECHWRITER</td>
</tr>
<tr>
<td>20</td>
<td>ADMIN</td>
</tr>
<tr>
<td>30</td>
<td>HR</td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>
```

Map-side Join: Parallel Scans

- How can we accomplish this in parallel?
  - Partition and sort both relations in the same manner

  - In MapReduce:
    - Map over one relation, read from other corresponding partition
    - No reducers necessary (unless to repartition or resort)

  - Consistently partitioned relations: realistic to expect?
    - Depends on the workflow
    - For ad hoc data analysis, reduce-side are more general, although less efficient

```
<table>
<thead>
<tr>
<th>DEPT</th>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>GALLOWAY</td>
</tr>
<tr>
<td>20</td>
<td>DULLEY</td>
</tr>
<tr>
<td>20</td>
<td>TAYLOR</td>
</tr>
<tr>
<td>20</td>
<td>HATHAWAY</td>
</tr>
<tr>
<td>30</td>
<td>REITER</td>
</tr>
<tr>
<td>30</td>
<td>PREVATT</td>
</tr>
</tbody>
</table>
```
In-Memory Join: Variants

- Basic idea: load one dataset into memory, stream over other dataset
- Works if \( R \ll S \) and \( R \) fits into memory
- Called a “hash join” in database terminology

Hash Join

Broadcast/Replication Join
In-Memory Join: Variants

- MapReduce implementation
  - Distribute R to all nodes
  - Map over S, each mapper loads R in memory, hashed by join key
  - For every tuple in S, look up join key in R
  - No reducers, unless for regrouping or resorting tuples

- Downside: need to copy R to all mappers
  - Not so bad, since R is small

Which Join to Use?

- In-memory join > map-side join > reduce-side join
  - Why?

- Limitations of each?
  - In-memory join: memory
  - Map-side join: sort order and partitioning
  - Reduce-side join: general purpose algorithm but sensible to data skewness?

- What about non-equi joins?
  - Inequality (S.A < R.A): map just forwards R-tuples, but replicates S-tuples for all larger R.A values as keys
Problems With Standard Repartition Equi-Joins

- Degree of parallelism limited by number of distinct join values
- Data skew
  - If one join value dominates, reducer processing that key will become bottleneck
- Does not generalize to other joins

Standard Repartition Equi-Join Algorithm

- Consider only the pairs with the same join attribute values

<table>
<thead>
<tr>
<th></th>
<th>S_1</th>
<th>S_2</th>
<th>S_3</th>
<th>S_4</th>
<th>S_5</th>
<th>S_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>r_1</td>
<td>a_1</td>
<td>a_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_2</td>
<td>a_1</td>
<td>a_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_3</td>
<td>a_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_4</td>
<td>a_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Naïve join algorithm

Standard repartition join algorithm

<table>
<thead>
<tr>
<th></th>
<th>S_1</th>
<th>S_2</th>
<th>S_3</th>
<th>S_4</th>
<th>S_5</th>
<th>S_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>r_1</td>
<td>a_1</td>
<td>a_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_2</td>
<td>a_1</td>
<td>a_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_3</td>
<td>a_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_4</td>
<td>a_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reducer-Centric Cost Model

- Difference between join implementations starts with Map output

![Diagram showing the flow of data from mapper output to reducer, then to join output.]

Optimization Goal: Minimal Job Completion time

- Job completion time depends on the slowest map and reduce functions
- Balancing the workloads of map functions is easy and thus we ignore them
- Balance the workloads of reduce functions as evenly as possible
  - Assume all reducers are similarly capable
- Processing time at reducer is approximately monotonic in input and output size
- Hence need to minimize max-reducer-input or max-reducer-output
- Join problem classification
  - Input-size dominated: minimize max-reducer-input
  - Output-size dominated: minimize max-reducer-output
  - Input-output balanced: minimize combination of both

© Kyuseok Shim (VLDB 2012 TUTORIAL)
Join Model

- Join-matrix $M$: $M(i, j) = \text{true}$, if and only if $(s_i, t_j)$ in join result
- Cover each true-valued cell by exactly one reducer

Reduce Allocations for Repartition Equi-joins

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

© Kyuseok Shim (VLDB 2012 TUTORIAL)
Comparison of Reduce Allocation Methods

- Simple allocation
  - Minimize the maximum input size of reduce functions
  - Output size may be skewed
- Random allocation
  - Minimize the maximum output size of reduce functions
  - Input size may be increased due to duplication
- Balanced allocation
  - Minimize both maximum input and output sizes

How to Balance Reduce Allocation

- Assume $r$ is desired number of reduce functions
- Partition join-matrix $M$ into $r$ regions
- A map function sends each record in $R$ and $S$ to mapped regions
- A reduce function outputs all possible $(r,s)$ pairs satisfying the join predicates in its value-list
- Propose M-Bucket-I algorithm [Okcan Riedewald: SIGMOD 2011]
Processing Relational Data: Summary

- **MapReduce algorithms for processing relational data:**
  - Group by, sorting, partitioning are handled automatically by shuffle/sort in MapReduce
  - Selection, projection, and other computations (e.g., aggregation), are performed either in mapper or reducer

- **Complex operations require multiple MapReduce jobs**
  - Example: top ten URLs in terms of average time spent
  - Opportunities for automatic optimization

- **Multiple strategies for relational joins**

Join Implementations on MapReduce

- MapReduce join implementations
  - $\theta$-join
  - Equijoin
    - Repartition join
    - Semi-join
    - Broadcast join
  - Similarity join
    - Map-only join
    - Trojan join
  - Multiway join
    - Multiple MapReduce jobs
    - Replicated join

**References**

Evolving Roles for Relational Database and MapReduce

The Traditional Way: Bringing Data to Compute

1. Complex Architecture
   - Many special-purpose systems
   - Moving data around
   - No complete views

2. Cost of Analytics
   - Existing systems strained
   - No agility
   - "BI backlog"

3. Time to Data
   - Up-front modeling
   - Transforms slow
   - Transforms lose data

4. Missing Data
   - Leaving data behind
   - Risk and compliance
   - High cost of storage

Evolution from Apache Hadoop to the Enterprise Data Hub A. Awadallah Co-Founder & CTO of Cloudera 2014
Need for High-Level Languages

- Hadoop is great for large-data processing!
  - But writing Java programs for everything is **verbose** and **slow**
  - Analysts don’t want to (or can’t) write Java

- **Solution**: develop higher-level data processing languages
  - Hive: HQL is like SQL
  - Pig: Pig Latin is a bit like Perl
Hive and Pig

- **Hive**: data warehousing application in Hadoop
  - Query language is HQL, variant of SQL
  - Tables stored on HDFS as flat files
  - Developed by Facebook, now open source

- **Pig**: large-scale data processing system
  - Scripts are written in Pig Latin, a dataflow language
  - Developed by Yahoo!, now open source
  - Roughly 1/3 of all Yahoo! internal jobs

- **Common idea**:
  - Provide higher-level language to facilitate large-data processing
  - Higher-level language “compiles down” to Hadoop jobs

Hive: Example

- Hive looks similar to an SQL database

- Relational join on two tables:
  - Table of word counts from Shakespeare collection
  - Table of word counts from the bible

```
SELECT s.word, s.freq, k.freq FROM shakespeare s
JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
ORDER BY s.freq DESC LIMIT 10;
```

<table>
<thead>
<tr>
<th>Word</th>
<th>Shakespeare Frequency</th>
<th>Bible Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>25848</td>
<td>62394</td>
</tr>
<tr>
<td>I</td>
<td>23031</td>
<td>8854</td>
</tr>
<tr>
<td>and</td>
<td>19671</td>
<td>38985</td>
</tr>
<tr>
<td>to</td>
<td>18038</td>
<td>13526</td>
</tr>
<tr>
<td>of</td>
<td>16700</td>
<td>34654</td>
</tr>
<tr>
<td>a</td>
<td>14170</td>
<td>8057</td>
</tr>
<tr>
<td>you</td>
<td>12702</td>
<td>2720</td>
</tr>
<tr>
<td>my</td>
<td>11297</td>
<td>4135</td>
</tr>
<tr>
<td>in</td>
<td>10797</td>
<td>12445</td>
</tr>
<tr>
<td>is</td>
<td>8882</td>
<td>6884</td>
</tr>
</tbody>
</table>

Source: Material drawn from Cloudera training VM
Hive: Behind the Scenes

SELECT s.word, s.freq, k.freq FROM shakespeare s INNER JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1 ORDER BY s.freq DESC LIMIT 10;

(Abstract Syntax Tree)

STAGE DEPENDENCIES:
Stage 1 depends on stages: Stage 1
Stage 0 is a root stage

STAGE PLANS:
Stage 0
Fetch Operator
limit: 10

Stage 1
Map Reduce
Alias = MapOperator Tree
Input format: org.apache.hadoop.mapred.SequenceFileInputFormat
Output format: org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat
key expressions: 
  expr: _col0
  type: int
  sort order: +
tag: -1
  value expressions: 
  expr: _col0
  type: string
  expr: _col1
  type: int
  expr: _col2
  type: int

Reduce Operator Tree:
Join Operator
condition map: Inner Join 0 to 1
condition expressions:
  0 {VALUE._col0} {VALUE._col1}
  1 {VALUE._col0}
outputColumnNames: _col0, _col1, _col2

Filter Operator
predicate:
  expr: ((_col0 >= 1) and (_col2 >= 1))
type: boolean

Select Operator
expressions:
  expr: _col1
  type: string
  expr: _col0
  type: int
  expr: _col2
  type: int
outputColumnNames: _col0, _col1, _col2

File Output Operator
compressed: false
GlobalTableId: 0
table: 
  input format: org.apache.hadoop.mapred.TextInputFormat
  output format: org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat

Stage 2
Map Reduce
Alias = MapOperator Tree
Input format: org.apache.hadoop.mapred.SequenceFileInputFormat
Output format: org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat
key expressions: 
  expr: _col1
  type: int
  sort order: -
tag: -1
  value expressions: 
  expr: _col0
  type: string
  expr: _col1
  type: int
  expr: _col2
  type: int

Reduce Operator Tree:
Extract
Limit
File Output Operator
compressed: false
GlobalTableId: 0
table: 
  input format: org.apache.hadoop.mapred.TextInputFormat
  output format: org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat

(one or more of MapReduce jobs)
Pig: Example

- Task: Find the top 10 most visited pages in each category

**Visits**

<table>
<thead>
<tr>
<th>User</th>
<th>Url</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>cnn.com</td>
<td>8:00</td>
</tr>
<tr>
<td>Amy</td>
<td>bbc.com</td>
<td>10:00</td>
</tr>
<tr>
<td>Amy</td>
<td>flickr.com</td>
<td>10:05</td>
</tr>
<tr>
<td>Fred</td>
<td>cnn.com</td>
<td>12:00</td>
</tr>
</tbody>
</table>

**Url Info**

<table>
<thead>
<tr>
<th>Url</th>
<th>Category</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnn.com</td>
<td>News</td>
<td>0.9</td>
</tr>
<tr>
<td>bbc.com</td>
<td>News</td>
<td>0.8</td>
</tr>
<tr>
<td>flickr.com</td>
<td>Photos</td>
<td>0.7</td>
</tr>
<tr>
<td>espn.com</td>
<td>Sports</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Pig Query Plan
Pig Script

visits = load '/data/visits' as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(visits);
urlInfo = load '/data/urlInfo' as (url, category, pRank);
visitCounts = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts, 10);

store topUrls into '/data/topUrls';

Pig Query Plan
References

- CS9223 – Massive Data Analysis J. Freire & J. Simeon New York University Course 2013
- INFM 718G / CMSC 828G Data-Intensive Computing with MapReduce J. Lin University of Maryland 2013
- CS 6240: Parallel Data Processing in MapReduce Mirek Riedewald Northeastern University 2014
- Extreme Computing Stratis D. Viglas University of Edinburg 2014
- MapReduce Algorithms for Big Data Analysis Kyuseok Shim VLDB 2012 TUTORIAL

Taxonomy of Parallel Architectures

Scales to 1000s of computers

- a) shared nothing
- b) shared disc
- c) shared memory

Easiest to program, but $$
Unicore vs Multi-core Architectures

Unicore
- T1
- T2
- CPU
- Memory Hierarchy

Multicore
- T1
- T2
- Core 1
- Core 2
- Core 3
- Core 4
- Memory Hierarchy

Positioning Big Data

Density of Data Value
- Data Warehousing
- Operational Analytics
- Ad-hoc Deep Analytics
- The New Data

Data Volume - Terabytes

Operational systems

Variety

£/TB

© 2014 IBM Corporation