Relational Data Processing on MapReduce

http://www.csd.uoc.gr/~hy562
University of Crete
Peta-scale Data Analysis

12+ TBs of tweet data every day

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

30 billion RFID tags today (1.3B in 2005)

100s of millions of GPS enabled devices sold annually

2+ billion people on the Web by end 2011

4.6 billion camera phones worldwide

30 billion views/day

YouTube is the 2nd most used search engine next to Google

76 million smart meters in 2009… 200M by 2014

© 2014 IBM Corporation
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:**
- Multi-purpose: 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

**Shared-nothing architecture for parallel processing**

Hadoop v2.0 (YARN) architecture
## MapReduce vs Parallel DBMS

<table>
<thead>
<tr>
<th>Feature</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>

[Pavlo et al., SIGMOD 2009, Stonebraker et al., CACM 2010, …]
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?

ETL (Extract, Transform, Load)

What about here?

Hadoop here?
Why does this make sense?
ETL Bottleneck

- Reporting is often a nightly task:
  - ETL is often slow
    - processing 24 h of data may take longer than 24 h!
- Often, with noisy datasets, ETL is the analysis!
  - ETL necessarily involves brute-force data scans: L, then E and T?
- Using Hadoop:
  - 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!
MapReduce Algorithms for Processing Relational Data
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

- **set operations**
  - set union
  - set intersection
  - set difference
  - cartesian product

- **relational database specific operations**
  - selection
  - projection
  - join

- **set functions**
  - sum
  - avg
  - count
  - any
  - max
  - min

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

$\pi_S(R)$
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 $(t, t)$
      - *equivalent in Spark: filter()*
  - **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*
  - Semi-structured 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

- \( R(X,Y) \setminus S(Y,Z) \)
  - **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 $γ_{A,θ(B)}(R)$
  - The map operation prepares the grouping e.g., emit (url, time) pairs
  - 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

R1  S1
R2  S2
R3  S3
R4  S4

R  S
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
Reduce-side 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)

Shuffling and sorting over the network

Reducer 1, Reducer 2, Reducer N

Reduction phase

Mapper 1, Mapper 2, Mapper 3, Mapper M

HDFS stores data blocks (Replicas are not shown)
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
Map-side (in-memory) Join

Load one dataset into memory, stream over other dataset

Distribute the smaller relation to all nodes
Map-side (in-memory) Join

- 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
Reducer-Centric Cost Model

- Difference between join implementations starts with Map output
Join Implementations on MapReduce

MapReduce join implementations

- θ-join
  - Equijoin
    - Repartition join
    - Semi-join
    - Broadcast join
  - Map-only join
    - Trojan join

- Similarity join
- Multiway join
  - Multiple MapReduce jobs
  - Replicated join

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
Evolving Roles for Relational Database and MapReduce
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></th>
<th>the</th>
<th>I</th>
<th>and</th>
<th>to</th>
<th>of</th>
<th>a</th>
<th>you</th>
<th>my</th>
<th>in</th>
<th>is</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25848</td>
<td>23031</td>
<td>19671</td>
<td>18038</td>
<td>16700</td>
<td>14170</td>
<td>12702</td>
<td>11297</td>
<td>10797</td>
<td>8882</td>
</tr>
<tr>
<td></td>
<td>62394</td>
<td>8854</td>
<td>38985</td>
<td>13526</td>
<td>34654</td>
<td>8057</td>
<td>2720</td>
<td>4135</td>
<td>12445</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
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)

```
(TOK_QUERY (TOK_FROM (TOK_JOIN (TOK_TABREF shakespeare s) (TOK_TABREF bible k) (= (. (TOK_TABLE_OR_COL s) word) (. (TOK_TABLE_OR_COL k) word)))) (TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE)) (TOK_SELECT (TOK_SELEXPR (. (TOK_TABLE_OR_COL s) word)) (TOK_SELEXPR (. (TOK_TABLE_OR_COL s) freq)) (TOK_SELEXPR (. (TOK_TABLE_OR_COL k) freq))) (TOK_WHERE (AND (>= (. (TOK_TABLE_OR_COL s) freq) 1) (>= (. (TOK_TABLE_OR_COL k) freq) 1))) (TOK_ORDERBY (TOK_TABSORTCOLNAMEDESC (. (TOK_TABLE_OR_COL s) freq)))) (TOK_LIMIT 10)))

(one or more of MapReduce jobs)
Hive: Behind the Scenes

STAGE DEPENDENCIES:
Stage-1 is a root stage
Stage-2 depends on stages: Stage-1
Stage-0 is a root stage

STAGE PLANS:
Stage: Stage-1
Map Reduce
Alias -> Map Operator Tree:
s
TableScan
alias: s
Filter Operator
predicate:
expr: (freq >= 1)
type: boolean
Reduce Output Operator
key expressions:
expr: word
type: string
sort order: +
Map-reduce partition columns:
expr: word
type: string
tag: 0
value expressions:
expr: freq
type: int
expr: word
type: string

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: Stage-2
Map Reduce
Alias -> Map Operator Tree:
hdfs://localhost:8022/tmp/hive-training/364214370/10002
Reduce Output Operator
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.SequenceFileInputFormat
output format: org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat

Stage: Stage-0
Fetch Operator
limit: 10
Pig: Example

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

<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>

<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

Load Visits

Group by url

Foreach url generate count

Load Url Info

Join on url

Group by category

Foreach category generate top10(urls)
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

Load Visits → Map₁ → Group by url → Reduce₁ → Map₂

Foreach url generate count

Load Url Info

Join on url

Group by category

Foreach category generate top10(urls)
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