Distributed data processing on the Cloud – Lecture 6

Joins with MapReduce and MapReduce limitations

Satish Srirama

Some material adapted from slides by Jimmy Lin, 2015 (licensed under Creation Commons Attribution 3.0 License)
Outline

• Structured data
  – Processing relational data with MapReduce

• MapReduce limitations
Relational Databases

• A relational database is comprised of tables
• Each table represents a relation = collection of tuples (rows)
• Each tuple consists of multiple fields (columns)
Basic SQL Commands - Queries

CREATE DATABASE mydb;
USE mydb;
CREATE TABLE mytable ( id INT PRIMARY KEY, name VARCHAR(20) );
INSERT INTO mytable VALUES ( 1, 'Will' );
INSERT INTO mytable VALUES ( 2, 'Marry' );
INSERT INTO mytable VALUES ( 3, 'Dean' );
SELECT id, name FROM mytable WHERE id = 1;
UPDATE mytable SET name = 'Willy' WHERE id = 1;
SELECT id, name FROM mytable;
DELETE FROM mytable WHERE id = 1;
SELECT id, name FROM mytable;
DROP DATABASE mydb;

https://mariadb.com/kb/en/library/basic-sql-statements/
Design Pattern: Secondary Sorting

• MapReduce sorts input to reducers by key
  – Values are arbitrarily ordered
• What if want to sort value also?
  – E.g., \( k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r) \ldots \)
• Use case:
  – We are monitoring different sensors
  – Data: (timestamp, sensor id, sensor value)
  – Map outputs
    sensorid -> (timestamp, sValue)
  – How to see the activity of each sensor over time?
Secondary Sorting: Solutions

• Solution 1:
  – Buffer values in memory, then sort
  – Why is this a bad idea?

• Solution 2:
  – “Value-to-key conversion” design pattern: form composite intermediate key, $(k, v_1)$
  – 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

\[ (k, v_1) \rightarrow (v_1, r) \]
\[ (k, v_3) \rightarrow (v_3, r) \]
\[ (k, v_4) \rightarrow (v_4, r) \]
\[ (k, v_8) \rightarrow (v_8, r) \]

…

Values arrive in sorted order…

Process by preserving state across multiple keys

Remember to partition correctly!
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

• Primitives
  – Projection ($\pi$)
  – Selection ($\sigma$)
  – Cartesian product ($\times$)
  – Set union ($\cup$)
  – Set difference ($\setminus$)
  – Rename ($\rho$)

• Other operations
  – Join ($\bowtie$)
  – Group by... aggregation
  – ...
Projection

\[ \pi \begin{array}{c}
R_1 \\
R_2 \\
R_3 \\
R_4 \\
R_5 \\
\end{array} \rightarrow
\begin{array}{c}
R_1 \\
R_2 \\
R_3 \\
R_4 \\
R_5 \\
\end{array} \]
Projection in MapReduce

• Easy!
  – Map over tuples, emit new tuples with appropriate attributes
  – No reducers, 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
  – Take advantage of compression when available
  – Semistructured data? No problem!
Selection

\[ \sigma \]
Selection in MapReduce

• Easy!
  – Map over tuples, emit only tuples that meet criteria
  – No reducers, 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
  – Take advantage of compression when available
  – Semistructured data? No problem!
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:
  - Map over tuples, emit time, keyed by url
  - Framework automatically groups values by keys
  - Compute average in reducer
  - Optimize with combiners
Relational Joins

R₁ R₂ R₃ R₄

S₁ S₂ S₃ S₄

R₁ R₂ R₃ R₄

S₁ S₂ S₃ S₄

R₁ R₂ R₃ R₄

S₁ S₂ S₃ S₄
Working Scenario – visited again

• Two tables:
  – User demographics (userid, gender, age, income, etc.)
  – User page visits (userid, 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
  – ...
Types of Relationships

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

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

• Basic idea: group by join key
  – Map over both sets of tuples
  – Emit tuple as value with join key as the intermediate key
  – Execution framework brings together tuples sharing the same key
  – Perform actual join in reducer
  – Similar to a “sort-merge join” in database terminology

• 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?
Reduce-side Join: V-to-K Conversion

In reducer...

- New key encountered: hold in memory
- Cross with records from other set

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td></td>
</tr>
<tr>
<td>$S_3$</td>
<td></td>
</tr>
<tr>
<td>$S_9$</td>
<td></td>
</tr>
<tr>
<td>$R_4$</td>
<td></td>
</tr>
<tr>
<td>$S_3$</td>
<td></td>
</tr>
<tr>
<td>$S_7$</td>
<td></td>
</tr>
</tbody>
</table>
Reduce-side Join: many-to-many

In reducer…

Keys: $R_1, R_5, R_8, S_2, S_3, S_9$
Values: $V_1, V_2, V_3, V_4, V_5, V_6$

Hold in memory

Cross with records from other set

What's the problem?
Problems with Reduce-side join

• In summary many-to-many is still left with the scalability issue

• Another big problem:
  – The basic idea behind the reduce-side join is to repartition the two datasets by the join key
  – This requires shuffling both the datasets across the network
Map-side Join: Basic Idea

Assume two datasets are sorted by the join key:

A sequential scan through both datasets to join (called a “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 datasets

• How can we accomplish this in parallel?
  – Partition and sort both datasets in the same manner
  – Example:
    • Suppose R and S were both divided into ten files, partitioned in the same manner by the join key
    • We simply need to merge join the first file of R with the first file of S, the second file of R with second of S, etc.

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

• Consistently partitioned datasets: realistic to expect?
  – Yes. Mostly MapReduce jobs are part of a workflow...
In-Memory Join

• Basic idea: load one dataset into memory, stream over other dataset
  – Works if R << S and R fits into memory
  – Called a “hash join” in database terminology

• 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
In-Memory Join: Variants

• Striped variant:
  – What if R is too big to fit into memory?
  – Divide R into R₁, R₂, R₃, ... s.t. each Rₙ fits into memory
  – Perform in-memory join: ∀n, Rₙ ⋈ S
  – Take the union of all join results

• Memcached join:
  – Load R into memcached
  – Replace in-memory hash lookup with memcached lookup
  – Memcached capacity >> RAM of individual node
  – Memcached scales out with cluster
  – Memcached is fast (basically, speed of network)
  – Batch requests to balance the latency costs
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
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
  – Multiple strategies for relational joins

• Complex operations require multiple MapReduce jobs
  – Example: top ten URLs in terms of average time spent
  – Opportunities for automatic optimization
Issues with MapReduce

• Java verbosity
  – Writing low level MapReduce code is slow
  – Need a lot of expertise to optimize MapReduce code
  – Prototyping takes significant time and requires manual compilation
  – A lot of custom code is required
    • Even simple functions such as sum and avg, you need to write reduce methods
  – Hard to manage more complex MapReduce job chains
• Pig is a high level language on top of Hadoop MapReduce (Next Lecture)
  – Similar to declarative SQL
    • Easier to get started
  – In comparison to Hadoop MapReduce:
    • 5% of the code
    • 5% of the time
Adapting Algorithms to MapReduce

• Designed a classification on how the algorithms can be adapted to MapReduce [Srirama et al, FGCS 2012]
  – Algorithm $\rightarrow$ single MapReduce job
    • Monte Carlo, RSA breaking
  – Algorithm $\rightarrow$ $n$ MapReduce jobs
    • CLustering LARge Applications - CLARA
  – Each iteration in algorithm $\rightarrow$ single MapReduce job
    • K-medoids (Clustering)
  – Each iteration in algorithm $\rightarrow$ $n$ MapReduce jobs
    • Conjugate Gradient

• Applicable especially for Hadoop MapReduce

Issues with Hadoop MapReduce

- It is designed and suitable for:
  - Data processing tasks
  - Embarrassingly parallel tasks

- Has serious issues with iterative algorithms
  - Long "start up" and "clean up" times ~17 seconds
  - No way to keep important data in memory between MapReduce job executions
  - At each iteration, all data is read again from HDFS and written back to HDFS, at the end
  - Results in a significant overhead in every iteration
Alternative Approaches - 1

- Restructuring algorithms into non-iterative versions
  - Can we change class 3 & 4 algorithms into class 1 or 2?

- **Iterative k-medoid clustering algorithm in MapReduce**
  - **Map**
    - Find the closest medoid and assign the object to the cluster of the medoid
    - Input: (cluster id, object)
    - Output: (new cluster id, object)
  - **Reduce**
    - Find which object is the most central and assign it as the new medoid of the cluster
    - Input: (cluster id, (list of all objects in the cluster))
    - Output: (cluster id, new medoid)
Alternative Approaches – 1 - continued

• CLARA was designed to make PAM algorithm more scalable [Kaufman and Rousseeuw (1990)]
  – CLARA applies PAM on random samples of the original dataset

• CLARA in MapReduce
  – The algorithm can be reduced to 2 MapReduce Jobs

• MapReduce job I (Finding the Candidate sets)
  – Map: (Assign random key to each point)
    • Input: < key, point >
    • Output < random key, point >
  – Reduce: (Pick first $S$ points and use k-medoids on them)
    • Output < key, k-medoids($S$ points) >
      – result of k-medoids() is a set of k medoids
CLARA in MapReduce - continued

- **MapReduce job II** (Measuring the quality over whole data set)
  - Map: (Find each points distance from it's closest medoid, for each candidate set)
    - Input: < key, point >
    - Output: < candidate set key, distance >
      - $C$ different sums, one for each candidate set
    - Reduce: (Sum distances)

- Candidate set and respective medoids with minimum sum will decide the clusters
Alternative Approaches - 2

• Using alternative MapReduce frameworks that are designed to handle iterative algorithms
  – In memory processing
  – Spark *(Lecture 8)*
Alternative Approaches - 3

• Alternative distributed computing models such as Bulk Synchronous Parallel model [Valiant, 1990] [Jakovits et al, HPCS 2013]

• Available BSP frameworks:
  – Graph processing: jPregel & Giraph (Lecture 11)
This week in lab...

• You’ll try Joins in MapReduce
Next Lecture

- Higher level scripting languages for distributed data processing - Pig
References

• Data-Intensive Text Processing with MapReduce
  Authors: Jimmy Lin and Chris Dyer


THANK YOU

srirama@ut.ee
Issues with MapReduce

• Java verbosity
• Hadoop task startup time
• Not good for iterative algorithms

  – Writing low level MapReduce code is slow
  – Need a lot of expertise to optimize MapReduce code
  – Prototyping requires compilation
  – A lot of custom code is required
  – Hard to manage more complex MapReduce job chains