Data-Intensive Distributed Computing
CS 431/631 451/651 (Fall 2019)

Part 5: Analyzing Relational Data (2/3)
October 22, 2019

Ali Abedi

These slides are available at https://www.student.cs.uwaterloo.ca/~cs451

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details
external APIs

- Frontend
- Backend

users

- Frontend
- Backend

users

- Frontend
- Backend

OLTP database

ETL (Extract, Transform, and Load)

“Data Lake”

Data Warehouse

- Other tools
- SQL on Hadoop
- “Traditional” BI tools

data scientists
What’s the selling point of SQL-on-Hadoop?
Trade (a little?) performance for flexibility
SQL-on-Hadoop

Today: How all of this works...
Hive: Example

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>the</th>
<th>25848</th>
<th>62394</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</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>
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;

Hive: Behind the Scenes

(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
 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.TextInputFormat
    output format: org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat

Stage: Stage-0
Fetch Operator
limit: 10
MapReduce algorithms for processing relational data
Relational Algebra

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

Other Operations
- Join ($\bowtie$)
- Group by... aggregation
  ...

Selection

\[ \sigma \]

\( R_1 \)

\( R_2 \)

\( R_3 \)

\( R_4 \)

\( R_5 \)
Selection in MapReduce

Easy!
In mapper: process each tuple, only emit tuples that meet criteria
Can be pipelined with projection
No reducers necessary (unless to do something else)

Performance mostly limited by HDFS throughput
Speed of encoding/decoding tuples becomes important
Take advantage of compression when available
Semistructured data? No problem!
Projection

\[ \pi \]

\[
\begin{array}{c}
R_1 \\
R_2 \\
R_3 \\
R_4 \\
R_5 \\
\end{array}
\]

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

Easy!
In mapper: process each tuple, re-emit with only projected attributes
    Can be pipelined with selection
    No reducers necessary (unless to do something else)

Implementation detail: bookkeeping required
Need to keep track of attribute mappings after projection
    e.g., name was r[4], becomes r[1] after projection

Performance mostly limited by HDFS throughput
    Speed of encoding/decoding tuples becomes important
        Take advantage of compression when available
    Semistructured data? No problem!
Group by... Aggregation

Aggregation functions:
AVG, MAX, MIN, SUM, COUNT, ...

MapReduce implementation:
Map over dataset, emit tuples, keyed by group by attribute
Framework automatically groups values by group by attribute
Compute aggregation function in reducer
Optimize with combiners, in-mapper combining

You already know how to do this!
Combiner Design

Combiners and reducers share same method signature

Sometimes, reducers can serve as combiners

Often, not...

Remember: combiner are optional optimizations

Should not affect algorithm correctness

May be run 0, 1, or multiple times

Example: find average of integers associated with the same key

SELECT key, AVG(value) FROM r GROUP BY key;
Computing the Mean: Version 1

class Mapper {
  def map(key: Text, value: Int, context: Context) = {
    context.write(key, value)
  }
}

class Reducer {
  def reduce(key: Text, values: Iterable[Int], context: Context) {
    for (value <- values) {
      sum += value
      cnt += 1
    }
    context.write(key, sum/cnt)
  }
}
class Mapper {
    def map(key: Text, value: Int, context: Context) =
        context.write(key, value)
}

class Combiner {
    def reduce(key: Text, values: Iterable[Int], context: Context) = {
        for (value <- values) {
            sum += value
            cnt += 1
        }
        context.write(key, (sum, cnt))
    }
}

class Reducer {
    def reduce(key: Text, values: Iterable[Pair], context: Context) = {
        for (value <- values) {
            sum += value.left
            cnt += value.right
        }
        context.write(key, sum/cnt)
    }
}
class Mapper {
    def map(key: Text, value: Int, context: Context) =
        context.write(key, (value, 1))
}

class Combiner {
    def reduce(key: Text, values: Iterable[Pair], context: Context) = {
        for (value <- values) {
            sum += value.left
            cnt += value.right
        }
        context.write(key, (sum, cnt))
    }
}

class Reducer {
    def reduce(key: Text, values: Iterable[Pair], context: Context) = {
        for (value <- values) {
            sum += value.left
            cnt += value.right
        }
        context.write(key, sum/cnt)
    }
}
class Mapper {
  val sums = new HashMap()
  val counts = new HashMap()

def map(key: Text, value: Int, context: Context) = {
  sums(key) += value
  counts(key) += 1
}

def cleanup(context: Context) = {
  for (key <- counts) {
    context.write(key, (sums(key), counts(key)))
  }
}
}
Relational Joins
Relational Joins

(Relational Joins)

(More precisely, an inner join)
Types of Relationships

Many-to-Many

One-to-Many

One-to-One
Join Algorithms in MapReduce

Reduce-side join
aka repartition join
aka shuffle join

Map-side join
aka sort-merge join

Hash join
aka broadcast join
aka replicated join
Reduce-side Join
aka repartition join, shuffle join

Basic idea: group by join key
- Map over both datasets
- Emit tuple as value with join key as the intermediate key
- Execution framework brings together tuples sharing the same key
- Perform join in reducer

Two variants
- 1-to-1 joins
- 1-to-many and many-to-many joins
Reduce-side Join: 1-to-1

Map

Reduce

More precisely, an inner join: What about outer joins?
Reduce-side Join: 1-to-many

Map

Reduce

What’s the problem?
Secondary Sorting

MapReduce sorts input to reducers by key
  Values may be arbitrarily ordered

What if we want to sort value also?
  E.g., \(k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\)...
Secondary Sorting: Solutions

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

Solution 2
“Value-to-key conversion” : form composite intermediate key, \((k, v_1)\)
Let the 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
\[ k \rightarrow (v_8, r_4), (v_1, r_1), (v_4, r_3), (v_3, r_2) \ldots \]

Values arrive in arbitrary order...

After
\[ (k, v_1) \rightarrow r_1 \]
\[ (k, v_3) \rightarrow r_2 \]
\[ (k, v_4) \rightarrow r_3 \]
\[ (k, v_8) \rightarrow r_4 \]

Values arrive in sorted order...

Process by preserving state across multiple keys

Remember to partition correctly!
Reduce-side Join: V-to-K Conversion

In reducer...

New key encountered: hold in memory

Cross with records from other dataset

New key encountered: hold in memory

Cross with records from other dataset
Reduce-side Join: many-to-many

In reducer...

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td></td>
</tr>
<tr>
<td>$R_5$</td>
<td></td>
</tr>
<tr>
<td>$R_8$</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>
</tbody>
</table>

Hold in memory

Cross with records from other dataset

What’s the problem?
Map-side Join
aka sort-merge join

Assume two datasets are sorted by the join key:

R₁ R₂ R₃ R₄
S₁ S₂ S₃ S₄

merge to join
Map-side Join
aka sort-merge join

Assume two datasets are sorted by the join key:

R_1  R_2  R_3  R_4
S_1  S_2  S_3  S_4

merge to join

merge to join

How can we parallelize this?  Co-partitioning
Map-side Join
aka sort-merge join

Works if...
Two datasets are co-partitioned
Sorted by join key

MapReduce implementation:
Map over one dataset, read from other corresponding partition
No reducers necessary (unless to do something else)

Co-partitioned, sorted datasets: realistic to expect?
Hash Join
aka broadcast join, replicated join

Basic idea:
Load one dataset into memory in a hashmap, keyed by join key
Read other dataset, probe for join key

Works if...
R << S and R fits into memory

MapReduce implementation:
Distribute R to all nodes (e.g., DistributedCache)
Map over S, each mapper loads R in memory and builds the hashmap
For every tuple in S, probe join key in R
No reducers necessary (unless to do something else)
Hash Join Variants

Co-partitioned variant:
R and S co-partitioned (but not sorted)?
Only need to build hashmap on the corresponding partition

Striped variant:
R too big to fit into memory?
Divide R into \( R_1, R_2, R_3, \ldots \) s.t. each \( R_n \) fits into memory
Perform hash join: \( \forall n, R_n \bowtie S \)
Take the union of all join results

Use a global key-value store:
Load R into memcached (or Redis)
Probe global key-value store for join key
Which join to use?

Hash join > map-side join > reduce-side join

Limitations of each?

In-memory join: memory
Map-side join: sort order and partitioning
Reduce-side join: general purpose
SQL-on-Hadoop

- SQL query interface
- Execution Layer
- HDFS
- Other Data Sources
Putting Everything Together

SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
    big2.f1 < 40 AND
    big2.f2 > 2;

Build logical plan
Optimize logical plan
Select physical plan

Note: generic SQL-on-Hadoop implementation; not exactly what Hive does, but pretty close.
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
    big2.f1 < 40 AND
    big2.f2 > 2;

Build logical plan
Optimize logical plan
Select physical plan
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
  big2.f1 < 40 AND
  big2.f2 > 2;

Build logical plan
Optimize logical plan
Select physical plan

Putting Everything Together
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
  big2.f1 < 40 AND
  big2.f2 > 2;

Build logical plan
Optimize logical plan
Select physical plan
Putting Everything Together

SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
  big2.f1 < 40 AND
  big2.f2 > 2;

Build logical plan
Optimize logical plan
Select physical plan
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
  big2.f1 < 40 AND
  big2.f2 > 2;
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
    big2.f1 < 40 AND
    big2.f2 > 2;

Putting Everything Together

Build logical plan
Optimize logical plan
Select physical plan
Hive: Behind the Scenes
Now you understand what’s going on here!

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;
Hive: Behind the Scenes

Now you understand what’s going on here!
SQL-on-Hadoop

SQL query interface

Execution Layer

HDFS

Other Data Sources
What about Spark SQL?

Based on the DataFrame API:
A distributed collection of data organized into named columns

Two ways of specifying SQL queries:

Directly:
```
val sqlContext = ... // An existing SQLContext
val df = sqlContext.sql("SELECT * FROM table")
// df is a dataframe, can be further manipulated...
```

Via DataFrame API:
```
// employees is a dataframe:
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```
Spark SQL: Query Planning

At the end of the day... it’s transformations on RDDs
Spark SQL: Physical Execution

Narrow Dependencies:
- map, filter
- union

Wide Dependencies:
- join with inputs co-partitioned
- groupByKey

= Map-side join

join with inputs not co-partitioned
= Reduce-side join

Hash join with broadcast variables
What’s the assignment?

SQL-on-Hadoop

SQL query interface

Spark

HDFS

You
What’s the assignment?

select
  l_returnflag,
  l_linenstatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
  sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from lineitem
where
  l_shipdate = 'YYYY-MM-DD'
group by l_returnflag, l_linenstatus;