Relational Algebra and MapReduce
Towards High-level Programming Languages

Pietro Michiardi

Eurecom

- Jimmy Lin and Chris Dyer, "Data-Intensive Text Processing with MapReduce," Morgan & Claypool Publishers, 2010. http://lintool.github.io/MapReduceAlgorithms/

- Tom White, "Hadoop, The Definitive Guide," O'Reilly / Yahoo Press, 2012

- Anand Rajaraman, Jeffrey D. Ullman, Jure Leskovec, "Mining of Massive Datasets", Cambridge University Press, 2013

Relational Algebra and MapReduce

Introduction

- ## Disclaimer
  - ▶ This is not a full course on Relational Algebra
  - ▶ Neither this is a course on SQL

- ## Introduction to Relational Algebra, RDBMS and SQL
  - ▶ Follow the video lectures of the Stanford class on RDBMS
    http://www.db-class.org/
  - → Note that you have to sign up for an account

- ## Overview of this part
  - ▶ Brief introduction to simplified relational algebra
  - ▶ Useful to understand Pig, Hive and HBase

Relational Algebra Operators

- **There are a number of operations on data that fit well the relational algebra model**
  - ▶ In traditional RDBMS, queries involve retrieval of small amounts of data
  - ▶ In this course, and in particular in this class, we should keep in mind the particular workload underlying MapReduce
  - → Full scans of large amounts of data
  - → Queries are not selective[1], they process all data

- **A review of some terminology**
  - ▶ A *relation* is a table
  - ▶ *Attributes* are the column headers of the table
  - ▶ The set of attributes of a relation is called a *schema*
    Example: $R(A_1, A_2, ..., A_n)$ indicates a relation called R whose attributes are $A_1, A_2, ..., A_n$

[1]This is true in general. However, most ETL jobs involve selection and projection to do data preparation.

Operators

- **Let's start with an example**
  - ▶ Below, we have part of a relation called *Links* describing the structure of the Web
  - ▶ There are two *attributes*: *From* and *To*
  - ▶ A row, or *tuple*, of the relation is a pair of URLs, indicating the existence of a link between them
  - $\rightarrow$ The number of tuples in a real dataset is in the order of billions (10^9)

From	To
url1	url2
url1	url3
url2	url3
url2	url4
...	...

Operators

- **Relations (however big) can be stored in a distributed filesystem**
  - If they don't fit in a single machine, they're broken into pieces (think HDFS)

- **Next, we review and describe a set of relational algebra operators**
  - Intuitive explanation of what they do
  - "Pseudo-code" of their implementation in/by MapReduce

Operators

- **Selection:** $\sigma_C(R)$
    - ▸ Apply condition *C* to each tuple of relation *R*
    - ▸ Produce in output a relation containing only tuples that satisfy *C*

- **Projection:** $\pi_S(R)$
    - ▸ Given a *subset S* of relation *R* attributes
    - ▸ Produce in output a relation containing only tuples for the attributes in *S*

- **Union, Intersection and Difference**
    - ▸ Well known operators on sets
    - ▸ Apply to the set of tuples in two relations that have the same schema
    - ▸ Variations on the theme: work on *bags*

Operators

- **Natural join** $R \bowtie S$
  - ▶ Given two relations, *compare each pair of tuples*, one from each relation
  - ▶ If the tuples agree on all the attributes common to both schema → produce an output tuple that has components on each attribute
  - ▶ Otherwise produce nothing
  - ▶ *Join condition* can be on a subset of attributes

- **Let's work with an example**
  - ▶ Recall the *Links* relation from previous slides
  - ▶ `Query` (or data processing job): `find the paths of length two in the Web`

Join Example

- **Informally, to satisfy the query we must:**
  - find the triples of URLs in the form (u, v, w) such that there is a link from u to v and a link from v to w

- **Using the join operator**
  - Imagine we have two relations (with different schema), and let's try to apply the natural join operator
  - There are two copies of *Links*: $L_1(U_1, U_2)$ and $L_2(U_2, U_3)$
  - Let's compute $L_1 \bowtie L_2$
    - ★ For each tuple t_1 of L_1 and each tuple t_2 of L_2, see if their U_2 component are the same
    - ★ If yes, then produce a tuple in output, with the schema (U_1, U_2, U_3)

Join Example

- **What we have seen is called (to be precise) a self-join**
  - ▶ Question: How would you implement a self join in your favorite programming language?
  - ▶ Question: What is the time complexity of your algorithm?
  - ▶ Question: What is the space complexity of your algorithm?

- **To continue the example**
  - ▶ Say you are not interested in the entire two-hop path but just the start and end nodes
  - ▶ Then you do a projection and the notation would be: $\pi_{U_1, U_3}(L_1 \bowtie L_2)$

Operators

- **Grouping and Aggregation:** $\gamma_X(R)$
  - ▶ Given a relation R, partition its tuples according to their values in one set of attributes G
    - ★ The set G is called the grouping attributes
  - ▶ Then, for each group, aggregate the values in certain other attributes
    - ★ Aggregation functions: `SUM`, `COUNT`, `AVG`, `MIN`, `MAX`, ...

- **In the notation, X is a list of elements that can be:**
  - ▶ A grouping attribute
  - ▶ An expression $\theta(A)$, where θ is one of the (five) aggregation functions and A is an attribute NOT among the grouping attributes

Operators

- **Grouping and Aggregation:** $\gamma_X(R)$
    - ▶ The result of this operation is a relation with one tuple for each group
    - ▶ That tuple has a component for each of the grouping attributes, with the value common to tuples of that group
    - ▶ That tuple has another component for each aggregation, with the aggregate value for that group

- **Let's work with an example**
    - ▶ Imagine that a social-networking site has a relation
      `Friends(User, Friend)`
    - ▶ The tuples are pairs (a, b) such that b is a friend of a
    - ▶ `Query: compute the number of friends each member has`

Grouping and Aggregation Example

- **How to satisfy the query**

    $\gamma_{User,\text{COUNT}(Friend))}(Friends)$

  ▸ This operation groups all the tuples by the value in their frist component
  → There is one group for each user
  ▸ Then, for each group, it counts the number of friends

- **Some details**

  ▸ The COUNT operation applied to an attribute does not consider the values of that attribute
  ▸ In fact, it counts the number of tuples in the group
  ▸ In SQL, there is a "count distinct" operator that counts the number of different values

Computing Selection

- **In practice, selections do not need a full-blown MapReduce implementation**
  - ▶ They can be implemented in the map phase alone
  - ▶ Actually, they could also be implemented in the reduce portion

- **A MapReduce implementation of** $\sigma_C(R)$
  - **Map:**   ★ For each tuple t in R, check if t satisfies C
  - ★ If so, emit a key/value pair (t, t)
  - **Reduce:**   ★ Identity reducer
  - ★ Question: single or multiple reducers?

- **NOTE: the output is not exactly a relation**
  - ▶ WHY?

Computing Projections

- **Similar process to selection**
  - ▶ But, projection may cause same tuple to appear several times

- **A MapReduce implementation of $\pi_S(R)$**

  Map:
  - ⋆ 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')

  Reduce:
  - ⋆ For each key t' produced by any of the Map tasks, fetch $t', [t', \cdots, t']$
  - ⋆ Emit a key/value pair (t', t')

- **NOTE: the reduce operation is duplicate elimination**
  - ▶ This operation is associative and commutative, so it is possible to optimize MapReduce by using a `Combiner` in each mapper

Computing Unions

- **Suppose relations R and S have the same schema**
    - Map tasks will be assigned chunks from either R or S
    - Mappers don't do much, just pass by to reducers
    - Reducers do duplicate elimination

- **A MapReduce implementation of union**

  Map: [2]

    - For each tuple t in R or S, emit a key/value pair (t, t)

  Reduce:
    - For each key t there will be either one or two values
    - Emit (t, t) in either case

---

[2]Hadoop MapReduce supports reading multiple inputs.

Computing Intersections

- **Very similar to computing unions**
  - Suppose relations *R* and *S* have the same schema
  - The map function is the same (an identity mapper) as for union
  - The reduce function must produce a tuple only if both relations have that tuple

- **A MapReduce implementation of intersection**
  - **Map:**   ★ For each tuple *t* in *R* or *S*, emit a key/value pair (*t*, *t*)
  - **Reduce:** ★ If key *t* has value list [*t*, *t*] then emit the key/value pair (*t*, *t*)
  - ★ Otherwise, emit the key/value pair (*t*, NULL)

Computing difference

- **Assume we have two relations *R* and *S* with the same schema**
  - ▶ The only way a tuple *t* can appear in the output is if it is in *R* but not in *S*
  - ▶ The map function passes tuples from *R* and *S* to the reducer
  - ▶ NOTE: it must inform the reducer whether the tuple came from *R* or *S*

- **A MapReduce implementation of difference**
  - **Map:**   ⋆ For a tuple *t* in *R* emit a key/value pair $(t, 'R')$ and for a tuple *t* in *S*, emit a key/value pair $(t, 'S')$
  - **Reduce:**   ⋆ For each key *t*, do the following:
    - ⋆ If it is associated to $'R'$, then emit (t, t)
    - ⋆ If it is associated to $['R', 'S']$ or $['S', 'R']$, or $['S']$, emit the key/value pair (t, NULL)

Computing the natural Join

- **This topic is subject to continuous refinements**
  - ▶ There are many JOIN operators and many different implementations
  - ▶ We've seen some of them in the laboratory sessions

- **Let's look at two relations** $R(A, B)$ **and** $S(B, C)$
  - ▶ We must find tuples that agree on their B components
  - ▶ We shall use the B-value of tuples from either relation as the key
  - ▶ The value will be the other component and the name of the relation
  - ▶ That way the reducer knows from which relation each tuple is coming from

Computing the natural Join

- ### A MapReduce implementation of Natural Join

  Map:
  - ★ For each tuple (a, b) of R emit the key/value pair $(b, ('R', a))$
  - ★ For each tuple (b, c) of S emit the key/value pair $(b, ('S', c))$

  Reduce:
  - ★ Each key b will be associated to a list of pairs that are either $('R', a)$ or $('S', c)$
  - ★ Emit key/value pairs of the form
    $(b, [(a_1, b, c_1), (a_2, b, c_2), \cdots , (a_n, b, c_n)])$

- ### NOTES
  - ▶ Question: what if the MapReduce framework wouldn't implement the distributed (and sorted) group by?
  - ▶ In general, for n tuples in relation R and m tuples in relation S all with a common B-value, then we end up with nm tuples in the result
  - ▶ If all tuples of both relations have the same B-value, then we're computing the **Cartesian product**

Grouping and Aggregation in MapReduce

- **Let $R(A, B, C)$ be a relation to which we apply $\gamma_{A,\theta(B)}(R)$**

  ▸ The map operation prepares the grouping
  ▸ The grouping is done by the framework
  ▸ The reducer computes the aggregation
  ▸ Simplifying assumptions: one grouping attribute and one aggregation function

- **MapReduce implementation of $\gamma_{A,\theta(B)}(R)$[3]**

  Map: ★ For each tuple (a, b, c) emit the key/value pair (a, b)
  Reduce: ★ Each key a represents a group
  ★ Apply θ to the list $[b_1, b_2, \cdots, b_n]$
  ★ Emit the key/value pair (a, x) where $x = \theta([b_1, b_2, \cdots, b_n])$

---

[3]Note here that we are also projecting.