MapReduce for Data Warehouses

Data Warehouses: Hadoop and Relational Databases

- In an enterprise setting, a data warehouse serves as a vast repository of data, holding everything from sales transactions to product inventories.
- Data warehouses form a foundation for business intelligence applications designed to provide decision support.
- Traditionally, data warehouses have been implemented through relational databases, particularly those optimized for a specific workload known as *online analytical processing* (OLAP).
- A number of vendors offer parallel databases, but customers find that they often cannot cost-effectively scale to the crushing amounts of data an organization needs to deal with today. Hadoop (and Spark) is a scalable and low-cost alternative.
- Facebook developed Hive (an open source project), which allows the data warehouse stored in Hadoop to be accessed via SQL queries (that get transformed into MapReduce operations under the hood).
- However, in many cases Hadoop and RDBMS co-exist, feeding into one another.

Relational MapReduce Patterns

Basic relational algebra operations that form the basis for relational databases.

- Selection
- Projection
- Union
- Intersection
- Difference
- GroupBy and Aggregation
- Join

Here we will implement them using MapReduce. We will also show the corresponding SQL database queries.

Selection

```
select *
from  Table_T
where (predicate is true);
```

```
class Mapper
   method Map(rowkey key, tuple t)
      if t satisfies the predicate
         Emit(tuple t, null)
```

Projection

Projection takes a relation and a (possibly empty) list of attributes of that relation as input. It outputs a relation containing only the specified list of attributes with duplicate tuples removed.

```
select  distinct A1, A2, A3  // where {A1, A2, A3} is a subset
from     Table_T;  // of the attributes for the table
```

Projection is just a little bit more complex than selection, since we use a Reducer to eliminate possible duplicates.

```
class Mapper
   method Map(rowkey key, tuple t)
      tuple g = project(t) // extract required fields to tuple g
      Emit(tuple g, null)

class Reducer
   method Reduce(tuple t, array n)   // n is an array of nulls
      Emit(tuple t, null) //skip duplicates (how?)
```

Union

```
select * from Table_T1
Union
select * from Table_T2
```

Mappers are fed by all records of the two sets to be united.
Reducer is used to eliminate duplicates.

```
class Mapper
   method Map(rowkey key, tuple t)
      Emit(tuple t, null)

class Reducer
   method Reduce(tuple t, array n) // n is an array of
      Emit(tuple t, null)          // one or two nulls
```

Intersection

```
select * from Table_T1
Intersection
select * from Table_T2
```

Mappers are fed by all records of the two sets to be intersected. Reducer emits only records that occurred twice. It is possible only if both sets contain this record because the record includes primary key and can occur in one set only once.

```
class Mapper
    method Map(rowkey key, tuple t)
        Emit(tuple t, null)

class Reducer
    method Reduce(tuple t, array n) // n is an array of
        if n.size() = 2             // one or two nulls
            Emit(tuple t, null)
```

Difference

```
select * from Table_T1
Except
select * from Table_T2
```

Suppose we have two sets of records — R and S. We want to compute difference R - S. Mapper emits all tuples with a tag, which is a name of the set this record came from. Reducer emits only records that came from R but not from S.

```
class Mapper
   method Map(rowkey key, tuple t)
  //t.SetName is either 'R' or 'S'
      Emit(tuple t, string t.SetName)

//array n can be ['R'], ['S'], ['R','S'], or ['S','R']
class Reducer
   method Reduce(tuple t, array n)
      if n.size() = 1 and n[1] = 'R'
          Emit(tuple t, null)
```

Group By and Aggregation

```
select pub_id, type, avg(price), sum(total_sales)
from titles
group by pub_id, type
```

- Mapper extracts from each tuple values to group by fields and aggregate fields and emits them.
- Reducer receives values to be aggregated already grouped and calculates an aggregation function.
- Typical aggregation functions like sum or max can be calculated in a streaming fashion, hence don't require handling all values simultaneously. In some cases two phase MapReduce job may be required (similar to Unique Items Counting).

```
class Mapper
 method Map(null, tuple[value GroupBy, value AggregateBy, value ...])
   Emit(value GroupBy, value AggregateBy)

class Reducer
 method Reduce(value GroupBy, [v1, v2,...])
   Emit(value GroupBy, aggregate([v1, v2,...])) //aggregate(): sum, max
```

Join

- A *Join* is a means for combining fields from two tables by using values common to each. ANSI standard SQL specifies four types of JOIN: INNER, OUTER, LEFT, and RIGHT. As a special case, a table can JOIN to itself in a self-join.
- An example:

```
CREATE TABLE department (
 DepartmentID INT,
 DepartmentName VARCHAR(20));

CREATE TABLE employee (
 LastName VARCHAR(20),
 DepartmentID INT);

SELECT *
FROM employee
INNER JOIN department ON
employee.DepartmentID = department.DepartmentID;

--implicit join
SELECT *
FROM employee, department
WHERE employee.DepartmentID = department.DepartmentID;
```

More on Joins

- Relational databases are often *normalized* to eliminate duplication of information when objects may have one-to-many relationships.
- Joining two tables effectively creates another table which combines information from both tables. This is at some expense in terms of the time it takes to compute the join.
- While it is also possible to simply maintain a *denormalized* table if speed is important, duplicate information may take extra space, and add the expense and complexity of maintaining data integrity if data that is duplicated later changes.
- Three fundamental algorithms for performing a join operation are: *nested loop join*, *sort-merge join* and *hash join*.

Join Example

User Demographics.

- Let R represent a collection of user profiles, in which case the key value k could be interpreted as the primary key (e.g., user id). The tuples might contain demographic information such as age, gender, income, etc.
- The other data set, L, might represent logs of online activity. Each tuple might correspond to a page view of a particular URL and may contain additional information such as time spent on the page, ad revenue generated, etc. The key value k in these tuples could be interpreted as the foreign key that associates each individual page view with a user.
- Joining these two data sets would allow an analyst, for example, to break down online activity in terms of demographics.

Join Examples

Movie Data Mining. Suppose we have three tables shown below and we want to find the top ten movies (by average rating):

```
Ratings [UserID, MovieID, Rating, Timestamp]
Users   [UserID, Gender, Age, Occupation, Zip-code]
Movies  [MovieID, Title, Genres]
```

Team up with another student and figure out the query to get the answer.

Repartition Join

Join of two sets R and L on some key k.

- Mapper goes through all tuples from R and L, extracts key k from the tuples, marks tuple with a tag that indicates a set this tuple came from ('R' or 'L'), and emits tagged tuple using k as a key.
- Reducer receives all tuples for a particular key k and puts them into two buckets – for R and for L. When the two buckets are filled, Reducer runs nested loop over them and emits a cross join of the buckets.
- Each emitted tuple is a concatenation R-tuple, L-tuple, and key k.

```
class Mapper
 method Map(null, tuple [join_key k, value v1, value v2,...])
   Emit(join_key k, tagged_tuple [set_name tag, values [v1, v2, ...] ] )

class Reducer
 method Reduce(join_key k, tagged_tuples [t1, t2,...])
   H = new AssociativeArray : set_name -> values
   //separate values into 2 arrays
   for all tagged_tuple t in [t1, t2,...]
      H{t.tag}.add(t.values)

   //produce a cross-join of the two arrays
   for all values r in H{'R'}
      for all values l in H{'L'}
         Emit(null, [k, r, l] )
```

Replicated Join

- Let's assume that one of the two sets, say R, is relatively small. This is fairly typical in practice.
- If so, R can be distributed to all Mappers and each Mapper can load it and index by the join key (e.g. in a hash table). After this, Mapper goes through tuples of the set L and joins them with the corresponding tuples from R that are stored in the hash table.
- This approach is very effective because there is no need for sorting or transmission of the set L over the network. Also known as memory-backed join or simple hash join.

```
class Mapper
   method Initialize
      H = new AssociativeArray : join_key -> tuple from R
      R = loadR()
      for all [ join_key k, tuple [r1, r2,...] ] in R
         H{k} = H{k}.append( [r1, r2,...] )

   method Map(join_key k, tuple l)
      for all tuple r in H{k}
         Emit(null, tuple [k, r, l] )
```

Join Without Scalability Issues

Suppose we have two relations, generically named S and T.

$(k_1, s_1, \mathbf{S}_1)$
$(k_2, s_2, \mathbf{S}_2)$
$(k_3, s_3, \mathbf{S}_3)$
...

where k is the key we would like to join on, s_n is a unique id for the tuple, and the $\mathbf{S}_n$ after s_n denotes other attributes in the tuple (unimportant for the purposes of the join).

$(k_1, t_1, \mathbf{T}_1)$
$(k_3, t_2, \mathbf{T}_2)$
$(k_8, t_3, \mathbf{T}_3)$
...

where k is the join key, t_n is a unique id for the tuple, and the $\mathbf{T}_n$ after t_n denotes other attributes in the tuple.

Reduce-Side Join

- We map over both data sets and emit the join key as the intermediate key, and the tuple itself as the intermediate value. Since MapReduce guarantees that all values with the same key are brought together, all tuples will be grouped by the join key—which is exactly what we need to perform the join operation.
- This approach is known as a parallel sort-merge join in the database community.
- There are three cases to consider:
  - One to one.
  - One to many.
  - Many to many.

Reduce-side Join: one to one

- The reducer will be presented keys and lists of values along the lines of the following:
  $k_{23} \rightarrow [(s_{64}, \mathbf{S}_{64}), (t_{84}, \mathbf{T}_{84})]$
  $k_{37} \rightarrow [(s_{68}, \mathbf{S}_{68})]$
  $k_{59} \rightarrow [(t_{97}, \mathbf{T}_{97}), (s_{81}, \mathbf{S}_{81})]$
  $k_{61} \rightarrow [(t_{99}, \mathbf{T}_{99})]$
  ...
- If there are two values associated with a key, then we know that one must be from S and the other must be from T. We can proceed to join the two tuples and perform additional computations (e.g., filter by some other attribute, compute aggregates, etc.).
- If there is only one value associated with a key, this means that no tuple in the other data set shares the join key, so the reducer does nothing.

Reduce-side Join: one to many

- In the mapper, we instead create a composite key consisting of the join key and the tuple id (from either S or T). Two additional changes are required:
  - First, we must define the sort order of the keys to first sort by the join key, and then sort all tuple ids from S before all tuple ids from T.
  - Second, we must define the partitioner to pay attention to only the join key, so that all composite keys with the same join key arrive at the same reducer.
- After applying the value-to-key conversion design pattern, the reducer will be presented with keys and values along the lines of the following:
  $(k_{82}, s_{105}) \rightarrow [(\mathbf{S}_{105})]$
  $(k_{82}, t_{98}) \rightarrow [(\mathbf{T}_{98})]$
  $(k_{82}, t_{101}) \rightarrow [(\mathbf{T}_{101})]$
  $(k_{82}, t_{137}) \rightarrow [(\mathbf{T}_{137})]$
  ...
- Whenever the reducer encounters a new join key, it is guaranteed that the associated value will be the relevant tuple from S. The reducer can hold this tuple in memory and then proceed to cross it with tuples from T in subsequent steps (until a new join key is encountered).
- Since the MapReduce execution framework performs the sorting, there is no need to buffer tuples (other than the single one from S). Thus, we have eliminated the scalability bottleneck.

Reduce-side Join: many to many

- All the tuples from S with the same join key will be encountered first, which the reducer can buffer in memory. As the reducer processes each tuple from T, it is crossed with all the tuples from S. Of course, we are assuming that the tuples from S (with the same join key) will fit into memory, which is a limitation of this algorithm (and why we want to control the sort order so that the smaller data set comes first).

Hive

- Hive was created at Facebook to make it possible for analysts with strong SQL skills (but meager Java programming skills) to run queries on the huge volumes of data that Facebook stored in HDFS.
- Today, Hive is a successful open source Apache project used by many organizations as a general-purpose, scalable data processing platform.
- Hive Query Language is very similar to SQL. The data scientist can write Hive queries in the hive shell (or Hive web interface or via an API to a Hive server). The queries get converted into MapReduce operations and run on Hadoop HDFS file system.

Hadoop and Hive: A Local Case Study

- Simulated a business intelligence problem that was given to us by a local software company. Currently they solve the problem using a traditional database. If we were to store the data in HDFS, we could exploit the parallel processing capability of MapReduce/Hive.
- **Setup**. A $5000 database server with 8 cores and 8G of memory versus four commodity dual-core nodes with 2G of memory and worth about $1200 each for the HDFS cluster.
- **Results**. Increase data size until Hadoop/Hive outperforms the database solution.

Data size	Hadoop Speedup	Hive Speedup
3G	0.6	0.1
4G	2.6	0.4
10G	17.5	2.5
20G	23.9	3.8

- The Hive-based solution required no programming! The MapReduce solution did require programming but was simpler than traditional parallel programming.

HBase

- HBase is a distributed column-oriented database built on top of HDFS. HBase is the Hadoop application to use when you require real-time read/write random access to very large data sets.
- HBase is not relational and does not support SQL but it can support very large, sparsely populated tables on clusters made from commodity hardware.
- Integrated with Hadoop MapReduce for powerful access.

RDBMS Story

- Initial public launch.
  Move from local workstation to shared, remotely hosted MySQL instance with a well-defined schema.
- Service becomes more popular; too many reads hitting the database.
  Add *memcached* to cache common queries. Reads are now no longer strictly ACID (Atomicity, Consistency, Isolation, Durability); cached data must expire.
- Service continues to grow in popularity; too many writes hitting the database.
  Scale MySQL vertically by buying a beefed-up server with 64 cores, 256 GB of RAM, and banks of fast hard drives. Costly.
- New features increases query complexity; now we have too many joins.
  Denormalize your data to reduce joins. (That's not what they taught me in DBA school!)
- Rising popularity swamps the server; things are too slow.
  Stop doing any server-side computations.
- Some queries are still too slow.
  Periodically prematerialize the most complex queries, and try to stop joining in most cases.
- Reads are OK, but writes are getting slower and slower.
  Drop secondary indexes and triggers (no indexes?)

At this point, there are no clear solutions for how to solve your scaling problems. In any case, you'll need to begin to scale horizontally. You can attempt to build some type of partitioning on your largest tables, or look into expensive solutions that provide multiple master capabilities.

HBase Story

- No real indexes.
  Rows are stored sequentially, as are the columns within each row. Therefore, no issues with index bloat, and insert performance is independent of table size.
- Automatic partitioning.
  As your tables grow, they will automatically be split into regions and distributed across all available nodes.
- Scale linearly and automatically with new nodes.
  Add a node, point it to the existing cluster, and run the regionserver. Regions will automatically rebalance, and load will spread evenly.
- Commodity hardware.
  Clusters are built on $1,000–$5,000 nodes rather than $50,000 nodes. RDBMSs are I/O hungry, requiring more costly hardware. Easy for elastic scaling in the cloud.
- Fault tolerant.
  Lots of nodes means each is relatively insignificant. No need to worry about individual node downtime.
- Batch processing.
  MapReduce (and Spark) integration allows fully parallel, distributed jobs against your data with locality awareness.

If you stay up at night worrying about your database (uptime, scale, or speed), you would seriously consider making a jump from the RDBMS world to HBase (or another no-SQL distributed database)

References

- Jimmy Lin and Chris Dyer. Chapter 3 in *Data-Intensive Text Processing with MapReduce*.
- Ilya Katsov. *MapReduce Patterns, Algorithms, and Use Cases*. `http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/`
- Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. *MapReduce and parallel DBMSs: Friends or foes?* Communications of the ACM, 53(1):64–71, 2010.
- Jeffrey Dean and Sanjay Ghemawat. *MapReduce: A flexible data processing tool.* Communications of the ACM, 53(1):72–77, 2010.