Joins using MapReduce

Overview

As discussed before, a **Join** operation is a data processing operation that combines records from two sources (e.g., two different input files), putting together two records if their content satisfies a specified condition (called a *join condition*).

Join is a combination of a *cartesian product* operation, an operation that given two collections of records creates a collections of all possible pairs of records (one record from each input collection), and a *filtering/selection* operation that applies the join condition to each paired record in the result of the cartesian product.

As a result, one way to define/describe a join between two collections R and S on some join condition C based on the contents of records in R and S is through the following naïve algorithm:

```
Algorithm Join(R,S,C)

  for each t in R do
      for each s in S do
          if C is true for <t,s> then
             emit(<t,s>)
          end if
      end for
  end for
```

Issues With Implementing Joins in Distributed Computing Frameworks

The double-nested loop in the algorithm above is a *global loop* - each record from one source is compared to each record from another source.

For some join conditions, worst-case output is the entire cartesian product (e.g, for a join condition `R.x != S.y` in a situation when there is no pair or records with a match between the values of `R.x` and `S.y`).

Traditional DBMS techniques for processing joins concentrate on optimizing data complexity of the join algorithms, i.e., optimizing the total number of disk

reads made in order to compute the result. This is not necessarily the right way to view join operations in distributed computing frameworks.

Joins and MapReduce

In order to implement joins in MapReduce framework we need the following:

- An ability to work with two source files within the same MapReduce process.

It turs out that MapReduce (and Hadoop) provides two different ways to do this, and each gives rise to a different way of implenting the join operation.

- *Multiple Input Files* handled by (possibly) multiple mappers: this approach can be used to set up a Reduce-side join.
- *Distributed Cache*: this approach can be used to set up a Map-side join.

Each type of implementation has its own limitations. The basic idea behind each approach is:

- **Reduce-side join**: Mappers are used to process two files "in parallel". The outputs of each mapper are synchronized on the output key, so that pairs of records that need to be compared wind up as input the the same `reduce()` instance. The actual checking of the join condition happens inside the `reduce()` function/method.
- **Map-side join**: One of the two files (the smaller one) is loaded into main memory of each node on the cluster. Mappers process the other file, and compare each record from that file to the records from the smaller file. Reducer is used as a pass-through.

We describe each type of MapReduce join approach.

Reduce-Side Join

Reduce-side join performs the actual joining of the records inside the `reduce()` method.

We describe the basic approach using a simple join condition $R.x == S.y$. We will discuss generalizing the reduce-side join below.

Basic Idea. The basic approach for reduce-side join can be described as follows:

- Use two mappers to process source files R and S.
- Use $R.x$ and $S.y$ as the keys in the output of the mapper.
- The latter will cause outputs of both mappers with the same key value to be combined in the same reducer instance.
- Reducer needs to output the pairs of records the originated from different sources.

The coding trick. In order to properly organize the `reduce()` method, we need to let `reduce()` know the origin of each record - whether it came from R or from S. We use a simple *coding* trick - we include, as part of the value the each `map()` function outputs, a simple code, that supplies to `reduce()` the origin of the record.

The pseudocode below shows how to organize reduce-side join following the principles expressed above.

```
function mapR(key, value)

// value has attribute "x" that will become our new key

   newKey = value.x
   newValue = {"source": 1,     // 1 means the records came from R
               "content": value}
   emit(newKey, newValue)

end //mapR


function mapS(key, value)

// value has attribute "y" that will become our new key


   newKey = value.y
   newValue = {"source": 2,       //2 mean the record came from S
               "content": value}
   emit(newKey, newValue)
end // mapS


//Reduce combines the output of both mappers

function reduce(key, List<Object> values)

 //join condition R.x == S.y

  rRecords = []
  sRecords = []

  for val in values do

     if val.source == 1 then        // val is from source R
        rRecords.append(val.content)

     if val.source == 2 then       // val is from source S
        sRecords.append(val.content)
  end for

  // we have split the input list into two sublists
  // rRecords are records that came from R
  // sRecords are records that came from S

     for r in rRecords do
        for s in sRecords do
            emit(key, (r,s))   // each pair of objects needs to be returned
        end for
     end for
```

Beyond Equijoins. If the join condition is `R.x1 == S.y1 and R.x2 == S.y2 and ... and R.xk = S.yk`, then the `map()` functions can be updated to make output keys `(x1,...,xk)` and `(y1,...,yk)` respectively.

For non-equijoin joins, things are more complex. There is a reduce-side join solution for the case when the attributes involved in the join condition have finitely many values. We will study this solution in detail when we discuss *matrix multiplication*. In a nutshell, the solution is

- When a value $R.x$ is observed in some record r, emit r for each value of the key that can be paired with x.

For example, let $R.x$ and $S.y$ have values $0, 1, \ldots, 10$, and let the join condition be $R.x > S.y$. Then given some value $R.x = i$, the record r will be joined with any record $s \in S$ where $s.x \in 1, \ldots, i-1$. So, we can write our `mapR()` function as

```
function mapR(key, value)
 // condition is R.x > S.y
  newKey = value.x
  newValue = {"source": 1,
              "content":value}

  for i = 1 to newKey-1 do
     emit(i, newValue)
  end for

end //mapR
```

(the second `map()` function uses the same idea).

Reduce-side Join: advantages and disadgantages

Advantages:

- Uses both `map` and `reduce` stages of MapReduce "as intended"
- Both input source files can be of arbitrary size.
- Works very well when there are relatively few records from each source to be joined.
- Works very well for equijoins and for cartesian-product-style joins with limited number of values ("cartesian-product-style" means that some activity needs to be performed for every, or almost every pair of key values).

Disadvantages:

- Does not work when join condition is over attributes with a lot of values, and is not an equijoin.
- Reduce step can become a bottleneck if there are very few key values (everything converges to running very few reduce instances).

Map-Side Join

The map-side join works by exploiting the idea of *sideloading* one source file into main memory of each compute node on the cluster.

The map-side join is essentially a distributed version of a classical DBMS one pass join algorithm that loads one source (relational table) into main memory, and then scans through the second source (relational table) one block of records at a time, searching, for each record in the second source, for matches with any of the records from the first source.

For map-side join

- we split the larger file, and send each split to a separate compute node
- we "sideload" the smaller file to each compute node's main memory, making it available for the mapper functions.
- The `map()` function performs a join of one record from the larger file with all records of the smaller file.
- The `reduce()` function serves as an essential "pass-through" and performs no distinct workload.

Distribusted Cache

The file "sideloading" in Hadoop is done using the notion of Distributed Cache.

Distributed Cache. In distributed computing frameworks, a distributed cache is any information that is stored in the main memory of every compute node and is available to use with any computation performed on the cluster. The key feature of distributed cache is that the content on all nodes **is the same**.

Distributed Cache in Hadoop. Hadoop takes a **do-it-yourself** approach to distributed cache. Essentially, a `org.apache.mapreduce.Job` calss has a `addCacheFile()` method that can be used to add a file on the HDFS system to the list of files to become distributed cache.

However, beyond allowing the instance of the `Job` class to pass the file name to the *mapper* class, Hadoop does nothing else. Everything else needs to be manufactured using the methods in the mapper class.

Hadoop defines three core methods in the `org.apache.hadoop.mapreduce.Mapper`: `setup()`, `map()`, and `cleanup()`.

The map-side join uses `setup()` and `map()`.

- `setup()` is used to load one of the data sources into the Distributed Cache. Hadoop allows the developer to choose how to store the data - essentially, distributed cache becomes an instance variable for the `Mapper` class, and thus is accessible from the `map()` method.
- `map()` is used to perform the actual join. The record from the second file is compared to the records in the distributed cache, and any matches are emitted.
- `cleanup()`

Map-Side Join Using Distributed Cache

The pseudocode for a map-side join for an equijoin with the `R.x == S.y` join condition looks as follows.

```
// assume S is the smaller file that gets stored in distributed cache

function setupMap(filename)
  file = open(filename)

  initialize dCache   // dCache is a mapper class instance variable
                      // that is a collection class

  for record in file do
      key = record.y
      value = record
      dCache.insert((key,value))
  end for

end //setupMap

// map processes key-value pairs from R

function map(key, value, dCache)   // we add dCache as a parameter for clarity


  newKey = value.x
  matches = dCache.find(newKey)    // assume find() returns all records that
                                   // are indexed under the value "newKey", i.e. value.x

  for rec in matches do           // output one joined record per match
     emit(newKey, (value, rec))
  end for
end // map


function reduce(key, List<Object> values)

  for val in values do
      emit(null, val)   // or emit(val, null)
                        // or emit(key, val) if you want results sorted by key

  end for

end //reduce
```

Map-side Join: advantages and disadvantages

Advantages.

- Works for **any** join condition.
- Both `map` and `reduce` are distributed.
- Implements a well-known join algorithm.

Disadvantages

- One source file **must** fit in main memory of the compute nodes. This is probably the biggest limitation on the use of map-side join.
- `reduce()` is distributed, but does not really do anything. In some cases it can be made to perform projection on each output record.