Chapter 6

Processing Relational Data

NOTE: This chapter represents a work in progress!

One popular application of Hadoop is data-warehousing. In an enterprise setting, a data warehouse serves as a vast repository of data, holding everything from sales transactions to product inventories. Typically, the data is relational in nature, but increasingly data warehouses are used to store semi-structured data (e.g., query logs) as well as unstructured data. Data warehouses form a foundation for business intelligence applications designed to provide decision support. It is widely believed that insights gained by mining historical, current, and prospective data can yield competitive advantages in the marketplace.

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. Parallel databases are often quite expensive—on the order of tens of thousands of dollars per terabyte of user data. Over the past few years, Hadoop has gained popularity as a platform for data-warehousing. Hammerbacher [68], for example, discussed Facebook's experiences with scaling up business intelligence applications with Oracle databases, which they ultimately abandoned in favor of a Hadoop-based solution developed in-house called Hive (which is now an open-source project). Pig [114] is a platform for massive data analytics built on Hadoop and capable of handling structured as well as semi-structured data. It was originally developed by Yahoo, but is now also an open-source project.

Given successful applications of Hadoop to data-warehousing and complex analytical queries that are prevalent in such an environment, it makes sense to examine MapReduce algorithms for manipulating relational data. This section focuses specifically on performing relational joins in MapReduce. We should stress here that even though Hadoop has been applied to process relational data, Hadoop *is not a database*. There is an ongoing debate between advocates of parallel databases and proponents of MapReduce regarding the merits of both approaches for OLAP-type workloads. Dewitt and Stonebraker, two

well-known figures in the database community, famously decried MapReduce as "a major step backwards" in a controversial blog post.[1] With colleagues, they ran a series of benchmarks that demonstrated the supposed superiority of column-oriented parallel databases over Hadoop [120, 144]. However, see Dean and Ghemawat's counterarguments [47] and recent attempts at hybrid architectures [1].

We shall refrain here from participating in this lively debate, and instead focus on discussing algorithms. From an application point of view, it is highly unlikely that an analyst interacting with a data warehouse will ever be called upon to write MapReduce programs (and indeed, Hadoop-based systems such as Hive and Pig present a much higher-level language for interacting with large amounts of data). Nevertheless, it is instructive to understand the algorithms that underlie basic relational operations.

6.1 Relational Joins

This section presents three different strategies for performing relational joins on two datasets (relations), generically named S and T. Let us suppose that relation S looks something like the following:

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

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). Similarly, suppose relation T looks something like this:

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

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.

To make this task more concrete, we present one realistic scenario: S might represent a collection of user profiles, in which case k could be interpreted as the primary key (i.e., user id). The tuples might contain demographic information such as age, gender, income, etc. The other dataset, T, 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 k in these tuples could be interpreted as the foreign key that associates each individual page view with a user. Joining these two datasets would allow an analyst, for example, to break down online activity in terms of demographics.

[1] `http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/`

Reduce-Side Join

The first approach to relational joins is what's known as a *reduce-side* join. The idea is quite simple: we map over both datasets 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 [134]. In more detail, there are three different cases to consider.

The first and simplest is a *one-to-one* join, where at most one tuple from S and one tuple from T share the same join key (but it may be the case that no tuple from S shares the join key with a tuple from T, or vice versa). In this case, the algorithm sketched above will work fine. The reducer will be presented keys and lists of values along the lines of the following:

$$
\begin{aligned}
&k_{23} \rightarrow [(s_{64}, \mathbf{S}_{64}), (t_{84}, \mathbf{T}_{84})] \\
&k_{37} \rightarrow [(s_{68}, \mathbf{S}_{68})] \\
&k_{59} \rightarrow [(t_{97}, \mathbf{T}_{97}), (s_{81}, \mathbf{S}_{81})] \\
&k_{61} \rightarrow [(t_{99}, \mathbf{T}_{99})] \\
&\ldots
\end{aligned}
$$

Since we've emitted the join key as the intermediate key, we can remove it from the value to save a bit of space.[2] 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. However, recall that in the basic MapReduce programming model, no guarantees are made about value ordering, so the first value might be from S or 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 dataset shares the join key, so the reducer does nothing.

Let us now consider the *one-to-many* join. Assume that tuples in S have unique join keys (i.e., k is the primary key in S), so that S is the "one" and T is the "many". The above algorithm will still work, but when processing each key in the reducer, we have no idea when the value corresponding to the tuple from S will be encountered, since values are arbitrarily ordered. The easiest solution is to buffer all values in memory, pick out the tuple from S, and then cross it with every tuple from T to perform the join. However, as we have seen several times already, this creates a scalability bottleneck since we may not have sufficient memory to hold all the tuples with the same join key.

This is a problem that requires a secondary sort, and the solution lies in the value-to-key conversion design pattern we just presented. In the mapper, instead of simply emitting the join key as the intermediate key, 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

[2] Not very important if the intermediate data is compressed.

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:

$$
\begin{aligned}
&(k_{82}, s_{105}) \rightarrow [(\mathbf{S}_{105})] \\
&(k_{82}, t_{98}) \rightarrow [(\mathbf{T}_{98})] \\
&(k_{82}, t_{101}) \rightarrow [(\mathbf{T}_{101})] \\
&(k_{82}, t_{137}) \rightarrow [(\mathbf{T}_{137})] \\
&\ldots
\end{aligned}
$$

Since both the join key and the tuple id are present in the intermediate key, we can remove them from the value to save a bit of space.[3] 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.

Finally, let us consider the *many-to-many* join case. Assuming that S is the smaller dataset, the above algorithm works as well. Consider what happens at the reducer:

$$
\begin{aligned}
&(k_{82}, s_{105}) \rightarrow [(\mathbf{S}_{105})] \\
&(k_{82}, s_{124}) \rightarrow [(\mathbf{S}_{124})] \\
&\ldots \\
&(k_{82}, t_{98}) \rightarrow [(\mathbf{T}_{98})] \\
&(k_{82}, t_{101}) \rightarrow [(\mathbf{T}_{101})] \\
&(k_{82}, t_{137}) \rightarrow [(\mathbf{T}_{137})] \\
&\ldots
\end{aligned}
$$

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 dataset comes first).

The basic idea behind the reduce-side join is to repartition the two datasets by the join key. The approach isn't particularly efficient since it requires shuffling both datasets across the network. This leads us to the *map-side* join.

[3] Once again, not very important if the intermediate data is compressed.

Map-Side Join

Suppose we have two datasets that are both sorted by the join key. We can perform a join by scanning through both datasets simultaneously—this is known as a merge join in the database community. We can parallelize this by partitioning and sorting *both* datasets in the same way. For example, suppose S and T were both divided into ten files, partitioned in the same manner by the join key. Further suppose that in each file, the tuples were sorted by the join key. In this case, we simply need to merge join the first file of S with the first file of T, the second file with S with the second file of T, etc. This can be accomplished in parallel, in the map phase of a MapReduce job—hence, a *map-side* join. In practice, we map over one of the datasets (the larger one) and inside the mapper read the corresponding part of the other dataset to perform the merge join.[4] No reducer is required, unless the programmer wishes to repartition the output or perform further processing.

A map-side join is far more efficient than a reduce-side join since there is no need to shuffle the datasets over the network. But is it realistic to expect that the stringent conditions required for map-side joins are satisfied? In many cases, *yes*. The reason is that relational joins happen within the broader context of a workflow, which may include multiple steps. Therefore, the datasets that are to be joined may be the output of previous processes (either MapReduce jobs or other code). If the workflow is known in advance and relatively static (both reasonable assumptions in a mature workflow), we can engineer the previous processes to generate output sorted and partitioned in a way that makes efficient map-side joins possible (in MapReduce, by using a custom partitioner and controlling the sort order of key-value pairs). For *ad hoc* data analysis, reduce-side joins are a more general, albeit less efficient, solution. Consider the case where datasets have multiple keys that one might wish to join on—then no matter how the data is organized, map-side joins will require repartitioning of the data. Alternatively, it is always possible to repartition a dataset using an identity mapper and reducer. But of course, this incurs the cost of shuffling data over the network.

There is a final restriction to bear in mind when using map-side joins with the Hadoop implementation of MapReduce. We assume here that the datasets to be joined were produced by previous MapReduce jobs, so this restriction applies to keys the reducers in those jobs may emit. Hadoop permits reducers to emit keys that are different from the input key whose values they are processing (that is, input and output keys need not be the same, nor even the same type).[5] However, if the output key of a reducer is different from the input key, then the output dataset from the reducer will not necessarily be partitioned in a manner consistent with the specified partitioner (because the partitioner applies to the *input* keys rather than the *output* keys). Since map-side joins depend on consistent partitioning and sorting of keys, the reducers used to

[4] Note that this almost always implies a non-local read.

[5] In contrast, recall from Section 2.2 that in Google's implementation, reducers' output keys must be exactly same as their input keys.

generate data that will participate in a later map-side join *must not* emit any key but the one they are currently processing.

Memory-Backed Join

In addition to the two previous approaches to joining relational data that leverage the MapReduce framework to bring together tuples that share a common join key, there is a family of approaches we call *memory-backed* joins based on random access probes. The simplest version is applicable when one of the two datasets completely fits in memory on each node. In this situation, we can load the smaller dataset into memory in every mapper, populating an associative array to facilitate random access to tuples based on the join key. The mapper initialization API hook (see Section 3.1) can be used for this purpose. Mappers are then applied to the other (larger) dataset, and for each input key-value pair, the mapper probes the in-memory dataset to see if there is a tuple with the same join key. If there is, the join is performed. This is known as a simple hash join by the database community [51].

What if neither dataset fits in memory? The simplest solution is to divide the smaller dataset, let's say S, into n partitions, such that $S = S_1 \cup S_2 \cup \ldots \cup S_n$. We can choose n so that each partition is small enough to fit in memory, and then run n memory-backed hash joins. This, of course, requires streaming through the other dataset n times.

There is an alternative approach to memory-backed joins for cases where neither datasets fit into memory. A distributed key-value store can be used to hold one dataset in memory across multiple machines while mapping over the other. The mappers would then query this distributed key-value store in parallel and perform joins if the join keys match.[6] The open-source caching system memcached can be used for exactly this purpose, and therefore we've dubbed this approach *memcached* join. For more information, this approach is detailed in a technical report [95].

[6] In order to achieve good performance in accessing distributed key-value stores, it is often necessary to batch queries before making synchronous requests (to amortize latency over many requests) or to rely on asynchronous requests.