2.3 Algorithms Using Map-Reduce

Map-reduce is not a solution to every problem, not even every problem that profitably can use many compute nodes operating in parallel. As we mentioned in Section 2.1.2, the entire distributed-file-system milieu makes sense only when files are very large and are rarely updated in place. Thus, we would not expect to use either a DFS or an implementation of map-reduce for managing on-line retail sales, even though a large on-line retailer such as Amazon.com uses thousands of compute nodes when processing requests over the Web. The reason is that the principal operations on Amazon data involve responding to searches for products, recording sales, and so on, processes that involve relatively little calculation and that change the database.[2] On the other hand, Amazon might use map-reduce to perform certain analytic queries on large amounts of data, such as finding for each user those users whose buying patterns were most similar.

The original purpose for which the Google implementation of map-reduce was created was to execute very large matrix-vector multiplications as are needed in the calculation of PageRank (See Chapter 5). We shall see that matrix-vector and matrix-matrix calculations fit nicely into the map-reduce

[2]Remember that even looking at a product you don't buy causes Amazon to remember that you looked at it.

style of computing. Another important class of operations that can use map-reduce effectively are the relational-algebra operations. We shall examine the map-reduce execution of these operations as well.

2.3.1 Matrix-Vector Multiplication by Map-Reduce

Suppose we have an $n \times n$ matrix M, whose element in row i and column j will be denoted m_{ij}. Suppose we also have a vector $\mathbf{v}$ of length n, whose jth element is v_j. Then the matrix-vector product is the vector $\mathbf{x}$ of length n, whose ith element x_i is given by

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

If $n = 100$, we do not want to use a DFS or map-reduce for this calculation. But this sort of calculation is at the heart of the ranking of Web pages that goes on at search engines, and there, n is in the tens of billions.[3] Let us first assume that n is large, but not so large that vector $\mathbf{v}$ cannot fit in main memory and thus be available to every Map task.

The matrix M and the vector $\mathbf{v}$ each will be stored in a file of the DFS. We assume that the row-column coordinates of each matrix element will be discoverable, either from its position in the file, or because it is stored with explicit coordinates, as a triple (i, j, m_{ij}). We also assume the position of element v_j in the vector $\mathbf{v}$ will be discoverable in the analogous way.

The Map Function: The Map function is written to apply to one element of M. However, if $\mathbf{v}$ is not already read into main memory at the compute node executing a Map task, then $\mathbf{v}$ is first read, in its entirety, and subsequently will be available to all applications of the Map function performed at this Map task. Each Map task will operate on a chunk of the matrix M. From each matrix element m_{ij} it produces the key-value pair $(i, m_{ij}v_j)$. Thus, all terms of the sum that make up the component x_i of the matrix-vector product will get the same key, i.

The Reduce Function: The Reduce function simply sums all the values associated with a given key i. The result will be a pair (i, x_i).

2.3.2 If the Vector v Cannot Fit in Main Memory

However, it is possible that the vector $\mathbf{v}$ is so large that it will not fit in its entirety in main memory. It is not required that $\mathbf{v}$ fit in main memory at a compute node, but if it does not then there will be a very large number of disk accesses as we move pieces of the vector into main memory to multiply components by elements of the matrix. Thus, as an alternative, we can divide

[3]The matrix is sparse, with on the average of 10 to 15 nonzero elements per row, since the matrix represents the links in the Web, with m_{ij} nonzero if and only if there is a link from page j to page i. Note that there is no way we could store a dense matrix whose side was 10^{10}, since it would have 10^{20} elements.

the matrix into vertical *stripes* of equal width and divide the vector into an equal number of horizontal stripes, of the same height. Our goal is to use enough stripes so that the portion of the vector in one stripe can fit conveniently into main memory at a compute node. Figure 2.4 suggests what the partition looks like if the matrix and vector are each divided into five stripes.

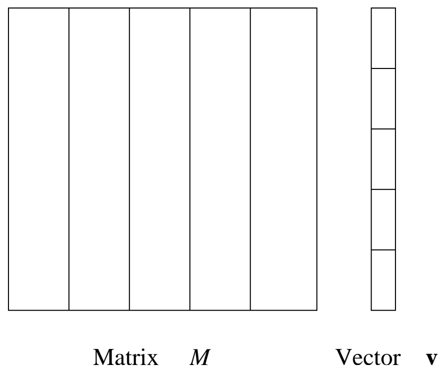

Figure 2.4: Division of a matrix and vector into five stripes

The ith stripe of the matrix multiplies only components from the ith stripe of the vector. Thus, we can divide the matrix into one file for each stripe, and do the same for the vector. Each Map task is assigned a chunk from one of the stripes of the matrix and gets the entire corresponding stripe of the vector. The Map and Reduce tasks can then act exactly as was described above for the case where Map tasks get the entire vector.

We shall take up matrix-vector multiplication using map-reduce again in Section 5.2. There, because of the particular application (PageRank calculation), we have an additional constraint that the result vector should be partitioned in the same way as the input vector, so the output may become the input for another iteration of the matrix-vector multiplication. We shall see there that the best strategy involves partitioning the matrix M into square blocks, rather than stripes.

2.3.3 Relational-Algebra Operations

There are a number of operations on large-scale data that are used in database queries. Many traditional database applications involve retrieval of small amounts of data, even though the database itself may be large. For example, a query may ask for the bank balance of one particular account. Such queries are not useful applications of map-reduce.

However, there are many operations on data that can be described easily in terms of the common database-query primitives, even if the queries themselves are not executed within a database management system. Thus, a good starting point for exploring applications of map-reduce is by considering the standard

operations on relations. We assume you are familiar with database systems, the query language SQL, and the relational model, but to review, a *relation* is a table with column headers called *attributes*. Rows of the relation are called *tuples*. The set of attributes of a relation is called its *schema*. We often write an expression like $R(A_1, A_2, \ldots, A_n)$ to say that the relation name is R and its attributes are A_1, $A_2, \ldots, A_n$.

From	*To*
url1	url2
url1	url3
url2	url3
url2	url4
···	···

Figure 2.5: Relation *Links* consists of the set of pairs of URL's, such that the first has one or more links to the second

Example 2.3 : In Fig. 2.5 we see part of the relation *Links* that describes the structure of the Web. There are two attributes, *From* and *To*. A row, or tuple, of the relation is a pair of URL's, such that there is at least one link from the first URL to the second. For instance, the first row of Fig. 2.5 is the pair $(url1, url2)$ that says the Web page $url1$ has a link to page $url2$. While we have shown only four tuples, the real relation of the Web, or the portion of it that would be stored by a typical search engine, has billions of tuples. □

A relation, however large, can be stored as a file in a distributed file system. The elements of this file are the tuples of the relation.

There are several standard operations on relations, often referred to as *relational algebra*, that are used to implement queries. The queries themselves usually are written in SQL. The relational-algebra operations we shall discuss are:

1. *Selection*: Apply a condition C to each tuple in the relation and produce as output only those tuples that satisfy C. The result of this selection is denoted $\sigma_C(R)$.

2. *Projection*: For some subset S of the attributes of the relation, produce from each tuple only the components for the attributes in S. The result of this projection is denoted $\pi_S(R)$.

3. *Union, Intersection*, and *Difference*: These well-known set operations apply to the sets of tuples in two relations that have the same schema. There are also bag (multiset) versions of the operations in SQL, with somewhat unintuitive definitions, but we shall not go into the bag versions of these operations here.

4. *Natural Join*: Given two relations, compare each pair of tuples, one from each relation. If the tuples agree on all the attributes that are common to the two schemas, then produce a tuple that has components for each of the attributes in either schema and agrees with the two tuples on each attribute. If the tuples disagree on one or more shared attributes, then produce nothing from this pair of tuples. The natural join of relations R and S is denoted $R \bowtie S$. While we shall discuss executing only the natural join with map-reduce, all *equijoins* (joins where the tuple-agreement condition involves equality of attributes from the two relations that do not necessarily have the same name) can be executed in the same manner. We shall give an illustration in Example 2.4.

5. *Grouping* and *Aggregation*:[4] Given a relation R, partition its tuples according to their values in one set of attributes G, called the *grouping attributes*. Then, for each group, aggregate the values in certain other attributes. The normally permitted aggregations are SUM, COUNT, AVG, MIN, and MAX, with the obvious meanings. Note that MIN and MAX require that the aggregated attributes have a type that can be compared, e.g., numbers or strings, while SUM and AVG require that the type allow arithmetic operations. We denote a grouping-and-aggregation operation on a relation R by $\gamma_X(R)$, where X is a list of elements that are either

   (a) A grouping attribute, or

   (b) An expression $\theta(A)$, where θ is one of the five aggregation operations such as SUM, and A is an attribute not among the grouping attributes.

   The result of this operation is 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. It also has a component for each aggregation, with the aggregated value for that group. We shall see an illustration in Example 2.5.

Example 2.4: Let us try to find the paths of length two in the Web, using the relation *Links* of Fig. 2.5. That is, we want to find the triples of URL's (u, v, w) such that there is a link from u to v and a link from v to w. We essentially want to take the natural join of *Links* with itself, but we first need to imagine that it is two relations, with different schemas, so we can describe the desired connection as a natural join. Thus, imagine that there are two copies of *Links*, namely $L1(U1, U2)$ and $L2(U2, U3)$. Now, if we compute $L1 \bowtie L2$, we shall have exactly what we want. That is, for each tuple $t1$ of $L1$ (i.e., each tuple of *Links*) and each tuple $t2$ of $L2$ (another tuple of *Links*, possibly even the same tuple), see if their $U2$ components are the same. Note that

[4] Some descriptions of relational algebra do not include these operations, and indeed they were not part of the original definition of this algebra. However, these operations are so important in SQL, that modern treatments of relational algebra include them.

these components are the second component of $t1$ and the first component of $t2$. If these two components agree, then produce a tuple for the result, with schema $(U1, U2, U3)$. This tuple consists of the first component of $t1$, the second component of $t1$ (which must equal the first component of $t2$), and the second component of $t2$.

We may not want the entire path of length two, but only want the pairs (u, w) of URL's such that there is at least one path from u to w of length two. If so, we can project out the middle components by computing $\pi_{U1,U3}(L1 \bowtie L2)$. □

Example 2.5 : Imagine that a social-networking site has a relation

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

This relation has tuples that are pairs (a, b) such that b is a friend of a. The site might want to develop statistics about the number of friends members have. Their first step would be to compute a count of the number of friends of each user. This operation can be done by grouping and aggregation, specifically

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

This operation groups all the tuples by the value in their first component, so there is one group for each user. Then, for each group the count of the number of friends of that user is made. The result will be one tuple for each group, and a typical tuple would look like (Sally, 300), if user "Sally" has 300 friends. □

2.3.4 Computing Selections by Map-Reduce

Selections really do not need the full power of map-reduce. They can be done most conveniently in the map portion alone, although they could also be done in the reduce portion alone. Here is a map-reduce implementation of selection $\sigma_C(R)$.

The Map Function: For each tuple t in R, test if it satisfies C. If so, produce the key-value pair (t, t). That is, both the key and value are t.

The Reduce Function: The Reduce function is the identity. It simply passes each key-value pair to the output.

Note that the output is not exactly a relation, because it has key-value pairs. However, a relation can be obtained by using only the value components (or only the key components) of the output.

2.3.5 Computing Projections by Map-Reduce

Projection is performed similarly to selection, because projection may cause the same tuple to appear several times, the Reduce function must eliminate duplicates. We may compute $\pi_S(R)$ as follows.

The Map Function: For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in S. Output the key-value pair (t', t').

The Reduce Function: For each key t' produced by any of the Map tasks, there will be one or more key-value pairs (t', t'). The Reduce function turns $(t', [t', t', \ldots, t'])$ into (t', t'), so it produces exactly one pair (t', t') for this key t'.

Observe that the Reduce operation is duplicate elimination. This operation is associative and commutative, so a combiner associated with each Map task can eliminate whatever duplicates are produced locally. However, the Reduce tasks are still needed to eliminate two identical tuples coming from different Map tasks.

2.3.6 Union, Intersection, and Difference by Map-Reduce

First, consider the union of two relations. Suppose relations R and S have the same schema. Map tasks will be assigned chunks from either R or S; it doesn't matter which. The Map tasks don't really do anything except pass their input tuples as key-value pairs to the Reduce tasks. The latter need only eliminate duplicates as for projection.

The Map Function: Turn each input tuple t into a key-value pair (t, t).

The Reduce Function: Associated with each key t there will be either one or two values. Produce output (t, t) in either case.

To compute the intersection, we can use the same Map function. However, the Reduce function must produce a tuple only if both relations have the tuple. If the key t has a list of two values $[t, t]$ associated with it, then the Reduce task for t should produce (t, t). However, if the value-list associated with key t is just $[t]$, then one of R and S is missing t, so we don't want to produce a tuple for the intersection.

The Map Function: Turn each tuple t into a key-value pair (t, t).

The Reduce Function: If key t has value list $[t, t]$, then produce (t, t). Otherwise, produce nothing.

The Difference $R - S$ requires a bit more thought. The only way a tuple t can appear in the output is if it is in R but not in S. The Map function can pass tuples from R and S through, but must inform the Reduce function whether the tuple came from R or S. We shall thus use the relation as the value associated with the key t. Here is a specification for the two functions.

The Map Function: For a tuple t in R, produce key-value pair (t, R), and for a tuple t in S, produce key-value pair (t, S). Note that the intent is that the value is the name of R or S (or better, a single bit indicating whether the relation is R or S), not the entire relation.

The Reduce Function: For each key t, if the associated value list is $[R]$, then produce (t, t). Otherwise, produce nothing.

2.3.7 Computing Natural Join by Map-Reduce

The idea behind implementing natural join via map-reduce can be seen if we look at the specific case of joining $R(A, B)$ with $S(B, C)$. We must find tuples that agree on their B components, that is the second component from tuples of R and the first component of tuples of S. 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, so the Reduce function can know where each tuple came from.

The Map Function: For each tuple (a, b) of R, produce the key-value pair $\big(b, (R, a)\big)$. For each tuple (b, c) of S, produce the key-value pair $\big(b, (S, c)\big)$.

The Reduce Function: Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c). Construct all pairs consisting of one with first component R and the other with first component S, say (R, a) and (S, c). The output from this key and value list is a sequence of key-value pairs. The key is irrelevant. Each value is one of the triples (a, b, c) such that (R, a) and (S, c) are on the input list of values.

The same algorithm works if the relations have more than two attributes. You can think of A as representing all those attributes in the schema of R but not S. B represents the attributes in both schemas, and C represents attributes only in the schema of S. The key for a tuple of R or S is the list of values in all the attributes that are in the schemas of both R and S. The value for a tuple of R is the name R together with the values of all the attributes belonging to R but not to S, and the value for a tuple of S is the name S together with the values of the attributes belonging to S but not R.

The Reduce function looks at all the key-value pairs with a given key and combines those values from R with those values of S in all possible ways. From each pairing, the tuple produced has the values from R, the key values, and the values from S.

2.3.8 Grouping and Aggregation by Map-Reduce

As with the join, we shall discuss the minimal example of grouping and aggregation, where there is one grouping attribute and one aggregation. Let $R(A, B, C)$ be a relation to which we apply the operator $\gamma_{A,\theta(B)}(R)$. Map will perform the grouping, while Reduce does the aggregation.

The Map Function: For each tuple (a, b, c) produce the key-value pair (a, b).

The Reduce Function: Each key a represents a group. Apply the aggregation operator θ to the list $[b_1, b_2, \ldots, b_n]$ of B-values associated with key a. The output is the pair (a, x), where x is the result of applying θ to the list. For example, if θ is SUM, then $x = b_1 + b_2 + \cdots + b_n$, and if θ is MAX, then x is the largest of b_1, $b_2, \ldots, b_n$.

If there are several grouping attributes, then the key is the list of the values of a tuple for all these attributes. If there is more than one aggregation, then

the Reduce function applies each of them to the list of values associated with a given key and produces a tuple consisting of the key, including components for all grouping attributes if there is more than one, followed by the results of each of the aggregations.

2.3.9 Matrix Multiplication

If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product $P = MN$ is the matrix P with element p_{ik} in row i and column k, where

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

It is required that the number of columns of M equals the number of rows of N, so the sum over j makes sense.

We can think of a matrix as a relation with three attributes: the row number, the column number, and the value in that row and column. Thus, we could view matrix M as a relation $M(I, J, V)$, with tuples (i, j, m_{ij}), and we could view matrix N as a relation $N(J, K, W)$, with tuples (j, k, n_{jk}). As large matrices are often sparse (mostly 0's), and since we can omit the tuples for matrix elements that are 0, this relational representation is often a very good one for a large matrix. However, it is possible that i, j, and k are implicit in the position of a matrix element in the file that represents it, rather than written explicitly with the element itself. In that case, the Map function will have to be designed to construct the I, J, and K components of tuples from the position of the data.

The product MN is almost a natural join followed by grouping and aggregation. That is, the natural join of $M(I, J, V)$ and $N(J, K, W)$, having only attribute J in common, would produce tuples (i, j, k, v, w) from each tuple (i, j, v) in M and tuple (j, k, w) in N. This five-component tuple represents the pair of matrix elements (m_{ij}, n_{jk}). What we want instead is the product of these elements, that is, the four-component tuple $(i, j, k, v \times w)$, because that represents the product $m_{ij} n_{jk}$. Once we have this relation as the result of one map-reduce operation, we can perform grouping and aggregation, with I and K as the grouping attributes and the sum of $V \times W$ as the aggregation. That is, we can implement matrix multiplication as the cascade of two map-reduce operations, as follows. First:

The Map Function: For each matrix element m_{ij}, produce the key value pair $\big(j, (M, i, m_{ij})\big)$. Likewise, for each matrix element n_{jk}, produce the key value pair $\big(j, (N, k, n_{jk})\big)$. Note that M and N in the values are not the matrices themselves. Rather they are names of the matrices or (as we mentioned for the similar Map function used for natural join) better, a bit indicating whether the element comes from M or N.

The Reduce Function: For each key j, examine its list of associated values. For each value that comes from M, say $(M, i, m_{ij})\big)$, and each value that comes

from N, say $\big(N, k, n_{jk})\big)$, produce a key-value pair with key equal to (i, k) and value equal to the product of these elements, $m_{ij}n_{jk}$.

Now, we perform a grouping and aggregation by another map-reduce operation.

The Map Function: This function is just the identity. That is, for every input element with key (i, k) and value v, produce exactly this key-value pair.

The Reduce Function: For each key (i, k), produce the sum of the list of values associated with this key. The result is a pair $\big((i, k), v\big)$, where v is the value of the element in row i and column k of the matrix $P = MN$.

2.3.10 Matrix Multiplication with One Map-Reduce Step

There often is more than one way to use map-reduce to solve a problem. You may wish to use only a single map-reduce pass to perform matrix multiplication $P = MN$. [5] It is possible to do so if we put more work into the two functions. Start by using the Map function to create the sets of matrix elements that are needed to compute each element of the answer P. Notice that an element of M or N contributes to many elements of the result, so one input element will be turned into many key-value pairs. The keys will be pairs (i, k), where i is a row of M and k is a column of N. Here is a synopsis of the Map and Reduce functions.

The Map Function: For each element m_{ij} of M, produce all the key-value pairs $\big((i, k),\ (M, j, m_{ij})\big)$ for $k = 1, 2, \ldots$, up to the number of columns of N. Similarly, for each element n_{jk} of N, produce all the key-value pairs $\big((i, k),\ (N, j, n_{jk})\big)$ for $i = 1, 2, \ldots$, up to the number of rows of M. As before, M and N are really bits to tell which of the two relations a value comes from.

The Reduce Function: Each key (i, k) will have an associated list with all the values (M, j, m_{ij}) and (N, j, n_{jk}), for all possible values of j. The Reduce function needs to connect the two values on the list that have the same value of j, for each j. An easy way to do this step is to sort by j the values that begin with M and sort by j the values that begin with N, in separate lists. The jth values on each list must have their third components, m_{ij} and n_{jk} extracted and multiplied. Then, these products are summed and the result is paired with (i, k) in the output of the Reduce function.

You may notice that if a row of the matrix M or a column of the matrix N is so large that it will not fit in main memory, then the Reduce tasks will be forced to use an external sort to order the values associated with a given key (i, k). However, in that case, the matrices themselves are so large, perhaps 10^{20} elements, that it is unlikely we would attempt this calculation if the matrices were dense. If they are sparse, then we would expect many fewer values to be associated with any one key, and it would be feasible to do the sum of products in main memory.

[5] However, we show in Section 2.6.7 that two passes of map-reduce are usually better than one for matrix multiplication.

2.3.11 Exercises for Section 2.3

Exercise 2.3.1 : Design map-reduce algorithms to take a very large file of integers and produce as output:

(a) The largest integer.

(b) The average of all the integers.

(c) The same set of integers, but with each integer appearing only once.

(d) The count of the number of distinct integers in the input.

Exercise 2.3.2 : Our formulation of matrix-vector multiplication assumed that the matrix M was square. Generalize the algorithm to the case where M is an r-by-c matrix for some number of rows r and columns c.

! **Exercise 2.3.3 :** In the form of relational algebra implemented in SQL, relations are not sets, but bags; that is, tuples are allowed to appear more than once. There are extended definitions of union, intersection, and difference for bags, which we shall define below. Write map-reduce algorithms for computing the following operations on bags R and S:

(a) *Bag Union*, defined to be the bag of tuples in which tuple t appears the sum of the numbers of times it appears in R and S.

(b) *Bag Intersection*, defined to be the bag of tuples in which tuple t appears the minimum of the numbers of times it appears in R and S.

(c) *Bag Difference*, defined to be the bag of tuples in which the number of times a tuple t appears is equal to the number of times it appears in R minus the number of times it appears in S. A tuple that appears more times in S than in R does not appear in the difference.

! **Exercise 2.3.4 :** Selection can also be performed on bags. Give a map-reduce implementation that produces the proper number of copies of each tuple t that passes the selection condition. That is, produce key-value pairs from which the correct result of the selection can be obtained easily from the values.

Exercise 2.3.5 : The relational-algebra operation $R(A, B) \bowtie_{B<C} S(C, D)$ produces all tuples (a, b, c, d) such that tuple (a, b) is in relation R, tuple (c, d) is in S, and $b < c$. Give a map-reduce implementation of this operation, assuming R and S are sets.