Optimizing Joins in a Map-Reduce Environment

Foto N. Afrati
National Technical University of Athens,Greece
afrati@softlab.ntua.gr

Jeffrey D. Ullman
Stanford University, USA
ullman@infolab.stanford.edu

ABSTRACT

Implementations of map-reduce are being used to perform many operations on very large data. We examine strategies for joining several relations in the map-reduce environment. Our new approach begins by identifying the "map-key," the set of attributes that identify the Reduce process to which a Map process must send a particular tuple. Each attribute of the map-key gets a "share," which is the number of buckets into which its values are hashed, to form a component of the identifier of a Reduce process. Relations have their tuples replicated in limited fashion, the degree of replication depending on the shares for those map-key attributes that are missing from their schema. We study the problem of optimizing the shares, given a fixed number of Reduce processes. An algorithm for detecting and fixing problems where an attribute is "mistakenly" included in the map-key is given. Then, we consider two important special cases: chain joins and star joins. In each case we are able to determine the map-key and determine the shares that yield the least replication. While the method we propose is not always superior to the conventional way of using map-reduce to implement joins, there are some important cases involving large-scale data where our method wins, including: (1) analytic queries in which a very large fact table is joined with smaller dimension tables, and (2) queries involving paths through graphs with high out-degree, such as the Web or a social network.

1. INTRODUCTION AND MOTIVATION

Search engines and other data-intensive applications have large amounts of data needing special-purpose computations. The canonical problem today is the sparse-matrix-vector calculation involved with PageRank [6], where the dimension of the matrix and vector can be in the 10's of billions. Most of these computations are conceptually simple, but their size has led implementors to distribute them across hundreds or thousands of low-end machines. This problem, and others like it, led to a new software stack to take the place of file systems, operating systems, and database-management systems.

Central to this stack is a file system such as the Google File System (GFS) [14] or Hadoop Distributed File System (HDFS) [2]. Such file systems are characterized by:

- Block (*chunk*) sizes that are perhaps 1000 times larger than those in conventional file systems — multimegabyte instead of multikilobyte.
- Replication of chunks in relatively independent locations (e.g., on different racks) to increase availability.

A powerful tool for building applications on such a file system is Google's map-reduce [10] or its open-source version, Hadoop [2]. Briefly, map-reduce allows a Map function to be applied to data stored in one or more files, resulting in key-value pairs. Many instantiations of the Map function can operate at once, and all their produced pairs are routed by a *master controller* to one of several Reduce processes, so that all pairs with the same key wind up at the same Reduce process. The Reduce processes apply another function to combine the values associated with one key to produce a single result for that key.

Map-reduce, inspired from functional programming, is a natural way to implement sparse-matrix-vector multiplication in parallel, and we shall soon see an example of how it can be used to compute parallel joins. Further, map-reduce offers resilience to hardware failures, which can be expected to occur during a massive calculation. The master controller manages Map and Reduce processes and is able to redo them if a process fails.

The new software stack includes higher-level, more database-like facilities, as well. Examples are Google's BigTable [7], or Yahoo!'s PNUTS [9], which can be thought of advanced file-level facilities. At a still higher level, Yahoo!'s PIG/PigLatin [19] translates relational operations such as joins into map-reduce computations. [8] suggests adding to map-reduce a "merge" phase and demonstrates how to express relationa-algebra operators thereby.

1.1 A Model for Cluster Computing

The same environment in which map-reduce proves so useful can also support interesting algorithms that do not fit the map-reduce form. Clustera [13] is an example of a system that allows more flexible programming than does Hadoop, in the same file environment. Although most of this paper is devoted to new algorithms that *do* fit the map-reduce framework, one could take advantage of more general computation plans (see Section 1.4). Here are the elements that describe

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

the environment in which computations like map-reduce can take place.

1. *Files*: A file is a set of tuples. It is stored in a file system such as GFS. That is, files are replicated with a very large chunk size. Unusual assumptions about files are:
   (a) We assume the order of tuples in a file cannot be predicted. Thus, these files are really relations as in a relational DBMS.
   (b) Many processes can read a file in parallel. That assumption is justified by the fact that all chunks are replicated and so several copies can be read at once.
   (c) Many processes can write pieces of a file at the same time. The justification is that tuples of the file can appear in any order, so several processes can write into the same buffer, or into several buffers, and thence into the file.
2. *Processes*: A process is the conventional unit of computation. It may obtain input from one or more files and write output to one or more files.
3. *Processors*: These are conventional nodes with a CPU, main memory, and secondary storage. We do not assume that the processors hold particular files or components of files. There is an essentially infinite supply of processors. Any process can be assigned to any one processor.

1.2 The Cost Measure for Algorithms

An *algorithm* in our model is an acyclic graph of processes with an arc from process P_1 to process P_2 if P_1 generates output that is (part of) the input to P_2. A process cannot begin until all of its input has been created. Note that we assume an infinite supply of processors, so any process can begin as soon as its input is ready.

- The *communication cost* of a process is the size of the input to this process. Note that we do not count the output size for a process. The output must be input to at least one other process (and will be counted there), unless it is output of the algorithm as a whole. We cannot do anything about the size of the result of an algorithm anyway. But more importantly, the algorithms we deal with are query implementations. The output of a query that is much larger than its input is not likely to be useful. Even analytic queries, while they may involve joining large relations, usually end by aggregating the output so it is meaningful to the user.
- The *total communication cost* is the sum of the communication costs of all processes comprising an algorithm.
- The *elapsed communication cost* is defined on the acyclic graph of processes. Consider a path through this graph, and sum the communication costs of the processes along that path. The maximum sum, over all paths, is the elapsed communication cost.

In our analysis, we do not account for the computation time taken by the processors. Typically, processing at a compute node can be done in main memory, if we are careful to assign limited amounts of work to each process. Thus, the cost of reading data from disk and shipping it over a network such as gigabit Ethernet will dominate the total elapsed time. Even in situations such as we shall explore, where a process involves joining several relations, we shall assume that tricks such as semijoins and judicious ordering can bring the processing cost down so it is at most commensurate with the cost of shipping data to the processor. The technique of Jakobsson [16] for chain joins, involving early duplicate elimination, would also be very important for multiway joins such as those that follow paths in the graph of the Web.

1.3 Outline of Paper and Our Contributions

In this paper, we investigate algorithms for taking joins of several relations in the environment just described. In particular, we are interested in algorithms that minimize the total communication cost. Our contributions are the following:

1. In Section 2, we begin the study of multiway (natural) joins. For comparison, we review the "normal" way to compute (2-way) joins using map-reduce. Through examples, we sketch an algorithm for multiway join evaluation that optimizes the communication cost by selecting properly those attributes that are used to partition and replicate the data among Reduce processes; the selected attributes form the *map-key*. We also show that there are some realistic situations in which the multiway join is more efficient than the conventional cascade of binary joins.
2. In Section 2.4 we introduce the notion of a "share" for each attribute of the map-key. The product of the shares is a fixed constant k, which is the number of Reduce processes used to implement the join. Each relation in a multiway join is replicated as many times as the product of the shares of the map-key attributes that are *not* in the schema for that relation.
3. The heart of the paper explores how to choose the map-key and shares to minimize the communication cost.
   - The method of "Lagrangean multipliers" lets us set up the communication-cost-optimization problem under the constraint that the product of the share variables is a constant k. There is an implicit constraint on the share variables that each must be a positive integer. However, optimization techniques such as Lagrange's do not support such constraints directly. Rather, they serve only to identify *points* (values for all the share variables) at which minima and maxima occur. Even if we postpone the matter of rounding or otherwise adjusting the share variables to be positive integers, we must still consider both minima that are identified by Lagrange's method by having all derivatives with respect to each of the share variables equal to 0, and points lying on the boundary of the region defined by requiring each share variable to be at least 1.

- In the common case, we simply set up the Lagrangean equations and solve them to find a minimum in the positive orthant (region with all share variables nonnegative). If some of the share variables are less than 1, we can set them to 1, their minimum possible value, and remove them from the map-key. We then re-solve the optimization problem for the smaller set of map-key attributes.
- Unfortunately, there are cases where the solution to the Lagrangean equations implies that at a minimum, one or more share variables are 0. What that actually means is that to attain a minimum in the positive orthant under the constraint of a fixed product of share variables, certain variables must approach 0, while other variables approach infinity, in a way that the product of all these variables remains a fixed constant. Section 3 explores this problem. We begin in Section 3.2 by identifying "dominated" attributes, which can be shown never to belong in a map-key, and which explain most of the cases where the Lagrangean yields no solution within the positive orthant.
- But dominated attributes in the map-key are not responsible for all such failures. Section 3.4 handles these rare but possible cases. We show that it is possible to remove attributes from the map-key until the remaining attributes allow us to solve the equations, although the process of selecting the right set of attributes to remove can be exponential in the number of attributes.
- Finally, in Section 3.5 we are able to put all of the above ideas together. We offer an algorithm for finding the optimal values of the share variables for any natural join.

4. Section 4 examines two common kinds of joins: chain joins and star joins (joins of a large fact table with several smaller dimension tables). For each of these types of joins we give closed-form solutions to the question of the optimal share of the map-key for each attribute.
   - In the case of star joins, the solution not only tells us how to compute the join in a map-reduce-type environment. It also suggests how one could optimize storage by partitioning the fact table permanently among all compute nodes and replicating each dimension table among a small subset of the compute nodes. This option is a realistic and easily adopted application of our techniques.

The complete version of this paper can be found in [1].

1.4 Joins and Map-Reduce

Multiway joins can be useful when processing large amounts of data as is the case in web applications. An example of a real problem that might be implemented in a map-reduce-like environment using multiway join is the HITS algorithm [17] for computing "hubs and authorities." While much of this paper is devoted to algorithms that can be implemented in the map-reduce framework, this problem can profit by going outside map-reduce, while still exploiting the computation environment in which map-reduce operates. In the full version we give details in support of the following claims:

- Multiway joins of very large data appear in practice.
- It is common for the results of these joins, although huge, to be aggregated so the output is somewhat compressed.
- Sometimes there are better ways to perform database queries in the cluster-computing environment than a sequence of map-reduce operations.

2. MULTIWAY JOINS

There is a straightforward way to join relations using map-reduce. We begin with a discussion of this algorithm. We then consider a different way to join several relations in one map-reduce operation.

2.1 The Two-Way Join and Map-Reduce

Suppose relations $R(A, B)$ and $S(B, C)$ are each stored in a file of the type described in Section 1.1. To join these relations, we must associate each tuple from either relation with a "key"[1] that is the value of its B-component. A collection of Map processes will turn each tuple (a, b) from R into a key-value pair with key b and value (a, R). Note that we include the relation with the value, so we can, in the Reduce phase, match only tuples from R with tuples from S, and not a pair of tuples from R or a pair of tuples from S. Similarly, we use a collection of Map processes to turn each tuple (b, c) from S into a key-value pair with key b and value (c, S). We include the relation name with the attribute value so in the reduce phase we only combine tuples from different relations.

The role of the Reduce processes is to combine tuples from R and S that have a common B-value. Thus, all tuples with a fixed B-value must be sent to the same Reduce process. Suppose we use k Reduce processes. Then choose a hash function h that maps B-values into k buckets, each hash value corresponding to one of the Reduce processes. Each Map process sends pairs with key b to the Reduce process for hash value $h(b)$. The Reduce processes write the joined tuples (a, b, c) that they find to a single output file.

2.2 Implementation Under Hadoop

If the above algorithm is implemented in Hadoop, then the partition of keys according to the hash function h can be done behind the scenes. That is, you tell Hadoop the value of k you desire, and it will create k Reduce processes and partition the keys among them using a hash function. Further, it passes the key-value pairs to a Reduce process with the keys in sorted order. Thus, it is possible to implement Reduce to take advantage of the fact that all tuples from R and S with a fixed value of B will appear consecutively on the input.

That feature is both good and bad. It allows a simpler implementation of Reduce, but the time spent by Hadoop in sorting the input to a Reduce process may be more than the time spent setting up the main-memory data structures that allow the Reduce processes to find all the tuples with a fixed value of B.

[1]Note that "keys" in the map-reduce sense are not unique. They are simply values used to distribute data between a Map process and the correct Reduce process.

2.3 Joining Several Relations at Once

Let us consider joining three relations

$$R(A,B) \bowtie S(B,C) \bowtie T(C,D)$$

We could implement this join by a sequence of two 2-way joins, choosing either to join R and S first, and then join T with the result, or to join S and T first and then join with R. Both joins can be implemented by map-reduce as described in Section 2.1.

An alternative algorithm involves joining all three relations at once, in a single map-reduce process. The Map processes send each tuple of R and T to many different Reduce processes, although each tuple of S is sent to only one Reduce process. The duplication of data increases the communication cost above the theoretical minimum, but in compensation, we do not have to communicate the result of the first join. As we shall see, the multiway join can therefore be preferable if the typical tuple of one relation joins with many tuples of another relation, as would be the case, for example, if we join copies of the matrix of the Web.

Much of this paper is devoted to optimizing the way this algorithm is implemented, but as an introduction, suppose we use $k = m^2$ Reduce processes for some m. Values of B and C will each be hashed to m buckets, and each Reduce process will be associated with a pair of buckets, one for B and one for C. That is, we choose to make B and C part of the map-key, and we give them equal shares.

Let h be a hash function with range $1, 2, \ldots, m$, and associate each Reduce process with a pair (i, j), where integers i and j are each between 1 and m. Each tuple $S(b, c)$ is sent to the Reduce process numbered $\big(h(b), h(c)\big)$. Each tuple $R(a, b)$ is sent to all Reduce processes numbered $\big(h(b), x\big)$, for any x. Each tuple $T(c, d)$ is sent to all Reduce processes numbered $\big(y, h(c)\big)$ for any y. Thus, each process (i, j) gets $1/m^2$th of S, and $1/m$th of R and T. An example, with $m = 4$, is shown in Fig. 1.

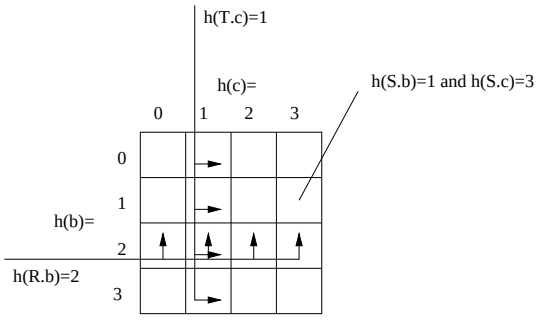


Figure 1: Distributing tuples of R, S, and T among $k = m^2$ processes

Each Reduce process computes the join of the tuples it receives. It is easy to observe that if there are three tuples $R(a, b)$, $S(b, c)$, and $T(c, d)$ that join, then they will all be sent to the Reduce process numbered $\big(h(b), h(c)\big)$. Thus, the algorithm computes the join correctly. Experiments were run to demonstrate some cases where the 3-way join is more efficient in practice (see full version).

2.4 An Introductory Optimization Example

In Section 2.3, we arbitrarily picked attributes B and C to form the map-key, and we chose to give B and C the same number of buckets, $m = \sqrt{k}$. This choice raises two questions:

1. Why are only B and C part of the map-key?
2. Is it best to give them the same number of buckets?

To learn how to optimize map-keys for a multiway join, let us begin with a simple example: the cyclic join

$$R(A,B) \bowtie S(B,C) \bowtie T(A,C)$$

Suppose that the target number of map-keys is k. That is, we shall use k Reduce processes to join tuples from the three relations. Each of the three attributes A, B, and C will have a *share* of the key, which we denote a, b, and c, respectively. We assume there are hash functions that map values of attribute A to a different buckets, values of B to b buckets, and values of C to c buckets. We use h as the hash function name, regardless of which attribute's value is being hashed. Note that $abc = k$.

- **Convention**: Throughout the paper, we use uppercase letters near the beginning of the alphabet for attributes and the corresponding lower-case letter as its share of a map-key. We refer to these variables $a, b, \ldots$ as *share variables*.

Consider tuples (x, y) in relation R. Which Reduce processes need to know about this tuple? Recall that each Reduce process is associated with a map-key (u, v, w), where u is a hash value in the range 1 to a, representing a bucket into which A-values are hashed. Similarly, v is a bucket in the range 1 to b representing a B-value, and w is a bucket in the range 1 to c representing a C-value. Tuple (x, y) from R can only be useful to this reducer if $h(x) = u$ and $h(y) = v$. However, it could be useful to any reducer that has these first two key components, regardless of the value of w. We conclude that (x, y) must be replicated and sent to the c different reducers corresponding to key values $\big(h(x), h(y), w\big)$, where $1 \leq w \leq c$.

Similar reasoning tells us that any tuple (y, z) from S must be sent to the a different reducers corresponding to map-keys $\big(u, h(y), h(z)\big)$, for $1 \leq u \leq a$. Finally, a tuple (x, z) from T is sent to the b different reducers corresponding to map-keys $\big(h(x), v, h(z)\big)$, for $1 \leq v \leq b$.

This replication of tuples has a communication cost associated with it. The number of tuples passed from the Map processes to the Reduce processes is

$$rc + sa + tb$$

where r, s, and t are the numbers of tuples in relations R, S, and T, respectively.

- **Convention**: We shall, in what follows, use R, S, $\ldots$ as relation names and use the corresponding lower-case letter as the size of the relation.

We must minimize the expression $rc+sa+tb$ subject to the constraint that $abc = k$. There is another constraint that we shall not deal with immediately, but which eventually must be faced: each of a, b, and c must be a positive integer. To

start, the method of Lagrangean multipliers serves us well. That is, we start with the expression

$$rc + sa + tb - \lambda(abc - k)$$

take derivatives with respect to the three variables, a, b, and c, and set the resulting expressions equal to 0. The result is three equations:

$$s = \lambda bc$$
$$t = \lambda ac$$
$$r = \lambda ab$$

These come from the derivatives with respect to a, b, and c in that order. If we multiply each equation by the variable missing from the right side (which is also the variable with respect to which we took the derivative to obtain that equation), and remember that abc equals the constant k, we get:

$$sa = \lambda k$$
$$tb = \lambda k$$
$$rc = \lambda k$$

We shall refer to equations derived this way (i.e., taking the derivative with respect to a variable, setting the result to 0, and then multiplying by the same variable) as the *Lagrangean equations*.

If we multiply the left sides of the three equations and set that equal to the product of the right sides, we get $rstk = \lambda^3 k^3$ (remembering that abc on the left equals k). We can now solve for $\lambda = \sqrt[3]{rst/k^2}$. From this, the first equation $sa = \lambda k$ yields $a = \sqrt[3]{krt/s^2}$. Similarly, the next two equations yield $b = \sqrt[3]{krs/t^2}$ and $c = \sqrt[3]{kst/r^2}$. When we substitute these values in the original expression to be optimized, $rc+sa+tb$, we get the minimum amount of communication between Map and Reduce processes: $3\sqrt[3]{krst}$.

Note that the values of a, b, and c are not necessarily integers. However, the values derived tell us approximately which integers the share variables need to be. They also tell us the desired ratios of the share variables; for example, $a/b = t/s$. In fact, the share variable for each attribute is inversely proportional to the size of the relation from whose schema the attribute is missing. This rule makes sense, as it says we should equalize the cost of distributing each of the relations to the Reduce processes. These ratios also let us pick good integer approximations to a, b, and c, as well as a value of k that is in the approximate range we want and is the product abc.

2.5 Comparison With Cascade of Joins

Under what circumstances is this 3-way join implemented by map-reduce a better choice than a cascade of two 2-way joins, each implemented by map-reduce. As usual, we shall not count the cost of producing the final result, since this result, if it is large, will likely be input to another operator such as aggregation, that reduces the size of the output.

To simplify the calculation, we shall assume that all three relations have the same size r. For example, they might each be the incidence matrix of the Web, and the cyclic query is asking for cycles of length 3 in the Web (this query might be useful, for example, in helping us identify certain kinds of spam farms).

If $r = s = t$, the communication between the Map and Reduce processes simplifies to $3r\sqrt[3]{k}$. We shall also assume that the probability of two tuples from different relations agreeing on their common attribute is p. For example, if the relations are incidence matrices of the Web, then rp equals the average out-degree of pages, which might be in the 10–15 range.

The communication of the optimal 3-way join is:

1. $3r$ for input to the Map processes.
2. $3r\sqrt[3]{k}$ for the input to the Reduce processes.

The second term dominates, so the total communication cost for the 3-way join is $O(r\sqrt[3]{k})$.

For the cascade of 2-way joins, whichever two we join first, we get an input size for the first Map processes of $2r$. This figure is also the input to the first Reduce processes, and the output size for the Reduce processes is r^2p. Thus, the second join's Map processes have an input size of r^2p for the intermediate join and r for the third relation. This figure is also the input size for the Reduce processes associated with the second join, and we do not count the size of the output from those processes. Assuming $rp > 1$, the r^2p term dominates, and the cascade of 2-way joins has total communication cost $O(r^2p)$.

We must thus compare r^2p with the cost of the 3-way join, which we found to be $O(r\sqrt[3]{k})$. That is, the 3-way join will be better as long as $\sqrt[3]{k}$ is less than rp. Since r and p are properties of the data, while k is a parameter of the join algorithm that we may choose, the conclusion of this analysis is that there is a limit on how large k can be in order for the 3-way join to be the method of choice. This limit is $k < (rp)^3$. For example, if $rp = 15$, as might be the case for the Web incidence matrix, then we can pick k up to 3375, and use that number of Reduce processes.

EXAMPLE 2.1. *Suppose $r = 10^7$, $p = 10^{-5}$, and $k = 1000$. Then the cost of the cascade of 2-way joins is $r^2p = 10^9$. The cost of the 3-way join is $r\sqrt[3]{k} = 10^8$, which is much less. Note also that the output size is small compared with both. Because there are three attributes that have to match to make a tuple in $R(A,B) \bowtie S(B,C) \bowtie T(A,C)$, the output size is $r^3p^3 = 10^6$.*

2.6 Trade-Off Between Speed and Cost

Before moving on to the general problem of optimizing multiway joins, let us observe that the example of Section 2.4 illustrates the trade-off that we face when using a method that replicates input. We saw that the total communication cost was $O(\sqrt[3]{krst})$. What is the elapsed communication cost?

First, there is no limit on the number of Map processes we can use, as long as each process gets at least one chunk of input. Thus, we can ignore the elapsed cost of the Map processes and concentrate on the k Reduce processes. Since the hash function used will divide the tuples of the relations randomly, we do not expect there to be much skew, except in some extreme cases. Thus, we can estimate the elapsed communication cost as $1/k$th of the total communication cost, or $O(\sqrt[3]{rst/k^2})$.

Thus, while the total cost grows as $k^{1/3}$, the elapsed cost shrinks as $k^{2/3}$. That is, the faster we want the join computed, the more resources we consume.

3. OPTIMIZATION OF MULTIWAY JOINS

Now, let us see how the example of Section 2.4 generalizes to arbitary natural joins. We shall again start out with an example that illustrates why certain attributes should not be allowed to have a share of the map-key. We then look at more complex situations where the Lagrangean equations do not have a feasible solution, and we show how it is possible to resolve those problems by eliminating attributes from the map-key.

3.1 A Preliminary Algorithm for Optimizing Share Variables

Here is an algorithm that generalizes the technique of Section 2.4. As we shall see, it sometimes yields a solution and sometimes not. Most of the rest of this section is devoted to fixing up the cases where it does not. Suppose that we want to compute the natural join of relations $R_1, R_2, \ldots, R_n$, and the attributes appearing among the relation schemas are $A_1, A_2, \ldots, A_m$.

Step 1: Start with the *cost expression*

$$\tau_1 + \tau_2 + \cdots + \tau_n - \lambda(a_1 a_2 \cdots a_m - k)$$

where τ_i is the term that represents the cost of communicating tuples of relations R_i to the Reduce processes that need the tuple. That is, τ_i is the product of r_i (the number of tuples in R_i) times the product of those share variables a_j such that attribute A_j does *not* appear in the schema of R_i. Note that this product of share variables is the number of Reduce processes to which each tuple of R_i must be distributed.

Step 2: For each share variable a_i, differentiate the cost expression with respect to a_i, and set the resulting expression to 0. Then, multiply the equation by a_i. The result is a collection of equations of the form $S_{a_i} = \lambda a_1 a_2 \cdots a_m$, where S_{a_i} is the sum of those τ_j's such that A_i is not in the schema of R_j. Since the product $a_1 a_2 \cdots a_m$ is constrained to equal k, we can write the equations as $S_{a_i} = \lambda k$. These are the "Lagrangean equations" for the join.

Step 3: Since Step 2 gives us m equations in m unknowns (the a_i's), we can in principle solve for the a_i's, in terms of λ, k, and the relations sizes, the r_i's. Including the equation that says the product of all the share variables is k, we can further eliminate λ.

3.2 Dominated Attributes

What can go wrong in Step 3? A lot. First, the solution for the share variables may assign some values less than 1. If so, we need to eliminate those share variables from the map-key and repeat the algorithm of Section 3.1 with the new map-key. However, there are more complex cases where the equations do not have a feasible solution because some of the τ_i's are forced to be 0. We shall study the simple case of this phenomenon, identify its cause (a "dominated" attribute), and show how to eliminate the problem. Then, in Section 3.4 we show how to deal with the general case where a sum of τ_i's is forced to be 0. We can work around these cases as well, although the algorithm to find a feasible solution can take time that is exponential in the number of attributes.

To see the simple case of the problem, as well as to illustrate the algorithm of Section 3.1, consider the following join:

$$R(A,B,C) \bowtie S(A,B,D) \bowtie T(A,D,E) \bowtie U(D,F)$$

whose hypergraph representation is shown in Fig. 2. In Step 1 we construct the cost expression:

$$rdef + scef + tbcf + uabce - \lambda(abcdef - k)$$

For example, the first term has the size of the relation R and the product of all share variables whose attributes are *not* in the schema of R.

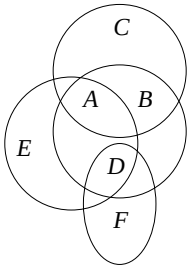


Figure 2: Hypergraph of relations illustrating dominated attributes

The Lagrangean equations of Step 2 are:

$$\begin{aligned} uabce &= \lambda k \\ tbcf + uabce &= \lambda k \\ scef + tbcf + uabce &= \lambda k \\ rdef &= \lambda k \\ rdef + scef + uabce &= \lambda k \\ rdef + scef + tbcf &= \lambda k \end{aligned}$$

Unfortunately, these equations do not yield a feasible solution. For example, if we subtract the first from the second, we get $tbcf = 0$. But since all share variables and relation sizes are positive integers, this situation is impossible. Several other terms can be shown equal to 0 as well.

A tool that lets us avoid some problems of this sort is the concept of "dominated attributes." Say attribute X *dominates* attribute Y if every schema that has Y also has X.

EXAMPLE 3.1. *For the four relations above, we see that A dominates B. That is, A appears in the schemas of R, S, and T, while B appears only in the schemas of R and S. However, B is not the only dominated attribute. In general, any attribute that appears only once in the join will be dominated. Thus, C, E, and F are also dominated.*

Any dominated attribute may be given a share equal to 1. Note that a variable with share 1 is effectively out of the map-key, since there is only one bucket for that attribute and all map-keys have the same value for that component.

3.3 Proof of the Dominator Rule

In proof, suppose we have a solution to some cost-minimization problem where attribute A dominates B, but the solution assigns a value $b > 1$ to the share for B. Replace a and b in the supposedly minimum solution by ab and 1, respectively. We claim that the cost of the solution does not increase. There are three possibilities regarding which of a and b is a factor of a term in the cost function:

1. Both a and b are factors. This case occurs for terms like $uabce$ in the running example of this section. The reason is that neither A nor B is an attribute of the schema of U. If both a and b are factors, their product is ab both before and after the transformation in the shares of A and B.

2. b is a factor, but a is not. This case corresponds to a relation that has A in its schema but not B, such as $tbcf$ in our running example. In this case, the cost function goes down when we replace b by 1.

3. Neither a nor b are factors. This case corresponds to a relation that has both A and B in its schema, such as $rdef$ in our running example. Here, changing a and/or b has no effect on the term.

The important point to observe that the fourth case, where a is a factor but b is not, cannot occur if A dominates B. Such a term corresponds to a relation whose schema has B but not A, and that is exactly what cannot occur if A dominates B.

3.4 Dealing With a Sum of Terms Equal to Zero

Whenever some sum of terms equals 0, we cannot get a feasible solution to the Lagrangean equations. Eliminating dominated attributes can handle some of these problems, as we saw in Section 3.2. However, more complex problems may sometimes arise.

In this section we present the general case of sums of terms equal to 0. We begin with a simple example where all relations have the same size; thus assume without loss of generality that $r = 1$. As we shall see, the argument we use does not really depend on the size of relations.

An Example

As in Section 3.1, denote by S_x the sum of terms in the communication cost function that have share variable x as a factor. Suppose for example that we have $S_a + S_e + S_c = 2S_b + S_d$. Suppose also that all terms on the right hand side (which is $2S_b + S_d$) of this equation are canceled by terms of the left hand side, so some terms in the left hand side remain and are equated to 0. A useful observation is that S_b contains only terms that have at least two of a, c, and e. The reason is that:

a) Each term in $S_a, S_b, \ldots$ has coefficient 1 (because all relations have the same size), and

b) Each term from $2S_b$ has to be canceled by either one term from S_a and one term from S_e or a term from S_a and a term from S_c or a term from S_c and a term from S_e.

A second useful observation is that S_d contains only terms that have at least one of a, c, and e.

Consider a solution a, c, e. If any of these variables are less than 1, the solution is not feasible. We must raise any fraction to 1, and can then eliminate that variable from the map-key. Assuming a, c, and e are all greater than 1, pick the smallest of them, say c. Do the following transformation: $b' = bc^2$; $d' = dc$; $a' = a/c$, $e' = e/c$, and $c' = 1$. All other share variables are unchanged. We claim that this transformation does not increase any term in the cost expression.

In proof, any term with b has at least two of a, c, and e. First, suppose the term does not also have d. There are four cases:

1. If the term has a and e but not c, then the new term has factor $b'a'e' = (bc^2)(a/c)(e/c) = bae$; i.e., the term does not change.

2. If the term has all of a, c, and e, then $b'a'c'e' = (bc^2)(a/c)(1)(e/c) = bae$, which is less than the original factor $bace$.

3. If the term has a and c, but not e, then
$$b'a'c' = (bc^2)(a/c)(1) = bac$$
i.e., the term does not change.

4. The final case, where the term has e and c but not a is similar.

For the terms with d but not b, there are seven cases, corresponding to the seven nonempty subsets of $\{a, c, e\}$ that may appear with d in the term. We shall consider only one; the others are analogous. Suppose the term has a, but not c or e. Then $d'a' = (dc)(a/c) = da$, so the term does not increase.

Finally, we must consider the case where both b and d appear in a term. In this case we argue that all of a, c, and e also appear in this term. In proof, the term appears on the right hand side $2 + 1 = 3$ times, so we must find this term three times on the left hand side as well. The only possibility is for this term to appear in all three S_x's on the left. Hence this term has factor $abcde$, and by the transformation remains the same.

The final step is to argue that the transformation does not violate the constraint that the product of all share variables is a given constant k. But that argument is the same as the last case above; the product $abcde$ does not change, and no other share variable was changed.

A Transformation That Eliminates a Sum of Terms Equal to Zero

In general, whenever there is an equality in which all the terms on one side also appear on the other side, then we can discover a transformation of the shares that sets one of the variables to 1. We can repeat this argument until all variables that can be eliminated are gone.

There is, however, a subtle but important point that must be addressed. In Example 3.4 we assumed that c was the smallest of a, c, and e. Yet we cannot tell which is smallest until we solve the equations, and until we get rid of all sums of terms equal to 0, we cannot solve the equations. Thus, we have to attempt solutions in which each of a, c, and e plays the role of the smallest of the three and take the best of what we get. Solving these three subcases may result in more share variables being eliminated by the same process, which in turn will multiply the number of subcases we need to solve. In the worst case, we have to solve a number of subproblems that is exponential in the number of share variables. However, the exponent is limited in general by the number of variables we are forced to set to 1.

Now we generalize the argument we used in the above example. We use the notation that we introduced in the

example above (S_a, etc.). By "sum of terms" we mean a sum of τ_i's (as in Section 3.1) with positive integer coefficients. The generalization is based on the following lemma:

LEMMA 3.2. *If there is a sum-of-terms = 0 then there are sums of terms S_{a_i} and S_{b_i} and positive integers m_i and n_i, such that the following hold:*

1. $\Sigma_{i=1}^{\mu} m_i S_{a_i} = \Sigma_{i=1}^{\nu} n_i S_{b_i}$, *and all the terms of the right hand side of the equation are canceled by terms of the left hand side.*
2. $\Sigma_{i=1}^{\mu} m_i = \Sigma_{i=1}^{\nu} n_i$

PROOF. (1) is simply a restatement of the fact that some sums and differences of the S_i's has led to a cancellation in which there are terms on one side and not the other.

(2) follows from the fact that each of the S_x's are equal, and equal to λk. If $\Sigma_{i=1}^{\mu} m_i \neq \Sigma_{i=1}^{\nu} n_i$, then we can replace all occurrences of S_i's by λk, and get that

$$(\Sigma_{i=1}^{\mu} m_i - \Sigma_{i=1}^{\nu} n_i)\lambda k = 0$$

But k is a chosen positive constant, and λ is a parameter that cannot be identically 0, so we would have a product of three nonzero values equal to 0. □

Convention:

- We call *conditionally optimal* a solution that minimizes the cost expression over real values ≥ 1 and under the constraint that the product of all shares is equal to a certain given number.
- We call *globally optimal* a solution that minimizes the cost expression over all real values and under the constraint that the product of all shares is equal to a certain given number.

Now we state (and prove in the complete version) the following theorem.

THEOREM 3.3. *Suppose there is a sum-of-terms = 0. Suppose that $a_0, b_0, \ldots$ is a conditionally optimal solution. Then there is a conditionally optimal solution $a'_0, b'_0, \ldots$ where one of the $a'_0, b'_0, \ldots$ is equal to 1.*

3.4.1 The Algorithm That Eliminates Sums of Terms Equal to 0

In conclusion, we proved above that the following algorithm handles the cases where there exists a sum of terms that equals zero. This algorithm takes as input a cost-minimization problem that possibly derives (using the Langragean method) equations with a positive linear combination equal to zero and produces a set of cost-minimization problems, none of which has a sum of terms equal to 0. The solution to the original optimization problem is the solution to that subproblem with the minimum optimized cost.

1. Suppose that for problem P, there is a sum of terms which is equal to zero. Then, let m_i, S_{a_i},n_i, and S_{b_i} be such that they satisfy the conditions in Lemma 3.2. For each a_i, replace a_i by 1 and create a new problem P_i with one fewer variable.
2. For each problem P_i we repeat the first step above if there is a sum of terms that equals to zero; otherwise, we go to Step 3 below.
3. We solve all the created subproblems above and, for each, we compute the optimum. The solution to our problem is the solution to the subproblem with the minimum optimum cost.

3.5 The Complete Algorithm

We can now describe an algorithm that yields the minimum-cost solution for apportioning the share variables among the attributes of a multiway natural join.

Step 1: Select those attributes that will get shares of the map-key. To do so, eliminate any attribute that is dominated by another attribute. In the case that two or more attributes appear in exactly the same schemas, eliminate all but one arbitrarily.

Step 2: Write the cost expression. This is the sum of one term for each relation in the join. The term for a relation R is the size r of that relation multiplied by the share variables for all the attributes that are in the map-key but not in the schema of R.

Step 3: Construct the Lagrangean equations for the join.

Step 4: Eliminate the cases where sum-of-terms = 0 according to the algorithm in subsection 3.4.1 and derive a set of subproblems to solve, in each of which there is no sum of terms being equal to zero.

Step 5: Find the conditionally optimal solution for each of the subproblems and keep the one with the minimum cost.

EXAMPLE 3.4. *Let us continue with the example*

$$R(A,B,C) \bowtie S(A,B,D) \bowtie T(A,D,E) \bowtie U(D,F)$$

that we started in Section 3.2. We have already established that Step 1 eliminates all attributes except A and D from the map-key. Thus, the cost expression for Step 2 is

$$rd + s + t + ua$$

In Step 3 we obtain the Lagrangean equations:

$$ua = \lambda k$$
$$rd = \lambda k$$

Since $ad = k$, we can multiply the two equations to get $ruad = ruk = \lambda^2 k^2$, or $ru = \lambda^2 k$. From this equality we deduce $\lambda = \sqrt{ru/k}$, then $a = \sqrt{rk/u}$ and $d = \sqrt{uk/r}$.

It would be nice if all equation sets were as easy to solve as those of Example 3.4, but unfortunately that does not appear to be the case. In the next section, we shall consider two important special cases — chain joins and star joins — and see that these cases are solvable in closed form.

In the complete version, we give the details of how to find a conditionally optimal solution in the cases where this does not coincide with a globally optimal solution.

3.6 Meaning of Solutions

Since we are solving nonlinear equations in general, we should not expect unique solutions. For example, notice in the simple 3-way join examined in Section 2.4, we developed one solution that had positive values for a, b, and c. However, if we negate any two of the three values, we get another solution that offers a lower communication cost. Of

course this solution is not in the feasible region, since all share variables must be 1 or more.

Even when we make the assumption that all values are positive, we often get a solution in which some share variables are less than 1. The cases discussed in Sections 3.2 and 3.4 result in certain variables being removed from the map-key, thus effectively forcing us to limit our search for solutions to a boundary of the feasible region, that is, to a subregion where certain variables are fixed at their lowest possible value, 1. While it makes intuitive sense to make this restriction and we have proved in the complete version that at least one optimal solution lies in this subregion, we have not ruled out the possibility of the existence of an optimal solution where one or more of these variables have larger values.

Finally, we have no guarantee that the solution we construct will have integer values. One might expect that rounding each noninteger to its nearest integer will offer the best integer solution, but there is no guarantee that is the case. We should observe that integer linear programs are often solved by finding the (noninteger) solution to the corresponding linear program and then rounding the fractions. However, that method is not guaranteed to produce the best integer solution.

Thus, we suggest that the solution we propose for optimizing multiway joins should be viewed as providing guidance as to which attributes deserve large shares and which do not. When deciding the exact shares we must deal not only with the constraint that shares be integers, but that their product must be the specific integer k. That further limits our choices to integers that evenly divide k, and in the rounding process we must preserve the product. An alternative approach to selecting shares is to treat k as a suggestion rather than a requirement. Then we can be more flexible in our selection of shares, as long as we choose integers that are near to the exact, noninteger values of the optimum solution.

4. IMPORTANT SPECIAL CASES

In this section, we consider the common case of a natural join that is a chain of relations, each linked to the following one by a single attribute. We prove a surprising simplification of the general problem: the terms of the cost expression always divide into two alternating groups with related values. Moreover, in the case of an even number of terms, one of these groups has values independent of the number of Reduce processes k. We begin with a study of star joins, where a fact table is joined with several dimension tables, and see that there is a simple solution in this case.

4.1 Star Joins

A star join has the form suggested by Fig. 3. A central *fact table*, represented here by the relation $ABCD$ is joined with several *dimension tables*, here represented by AE, BF, CG, and DH. It is expected that the fact table is very large, while the dimension tables are smaller. Moreover, the attribute or attributes shared by a dimension table and the fact table are normally a key for the dimension table. It is normal for there to be more attributes of the fact table than those shown, but these will not be part of the map-key for the join, and thus are not relevant to our discussion. Similarly, there may be more than one nonkey attribute of each fact table. Further, it is possible that there are several attributes shared between the fact table and one of the dimension tables, but for the purpose of optimizing the multiway join, we can combine them into one attribute as we have done in our example.

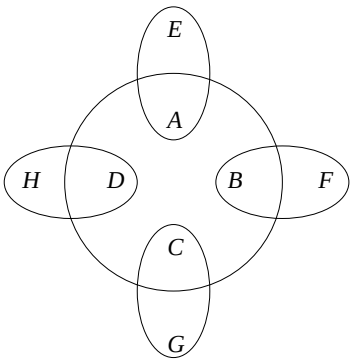


Figure 3: A star join

We shall generalize Fig. 3 to a fact table $F(A_1, A_2, \ldots, A_n)$ and n dimension tables, $D_i(A_i, B_i)$, for $i = 1, 2, \ldots, n$. First, observe that A_i dominates B_i, so we shall not have shares of the map-key for any of the B_i's. If we apply the method of Section 3.5, we start with the cost expression

$$f + kd_1/a_1 + kd_2/a_2 + \cdots + kd_n/a_n$$

where $k = a_1 a_2 \cdots a_n$, the product of all the shares. When we derive the equations for each of the share variables a_i, we find that the equation is missing the term f and the term kd_i/a_i but has all the other terms.

EXAMPLE 4.1. *Consider the join of Fig. 3. To name the relations, take the join to be*

$$R(A, B, C, D) \bowtie S(A, E) \bowtie T(B, F) \bowtie U(C, G) \bowtie V(D, H)$$

Then the cost expression is

$$r + sbcd + tacd + uabd + vabc$$

and the Lagrangean equations are:

$$\begin{aligned} tacd + uabd + vabc &= \lambda k \\ sbcd + uabd + vabc &= \lambda k \\ sbcd + tacd + vabc &= \lambda k \\ sbcd + tacd + uabd &= \lambda k \end{aligned}$$

If we subtract each equation from each other equation, we conclude that each of the four terms sbcd, tacd, uabd, and vabc must be equal. Remembering that abcd = k, we can write these four terms as $s/a = t/b = u/c = v/d$. Thus, the minimum-cost solution has shares for each variable proportional to the size of the dimension table in which it appears. That makes sense; it says that the map-keys partition the fact table into k parts, and each part of the fact table gets equal-sized pieces of each dimension table with which it is joined.

We can use these equations to solve for b, c, and d in terms of a. The result is $b = at/s$, $c = au/s$, and $d = av/s$. Then, using the fact that abcd = k, we derive $a^4tuv/s^3 = k$, or $a = \sqrt[4]{ks^3/tuv}$, $b = \sqrt[4]{kt^3/suv}$, $c = \sqrt[4]{ku^3/stv}$, and $d = \sqrt[4]{kv^3/stu}$.

We can easily generalize Example 4.1.

THEOREM 4.2. *Let $d = d_1 d_2 \cdots d_n$, that is, the product of the sizes of all the dimension tables. Then a_i, the share for the attribute that appears in the fact table's schema and the schema of the ith dimension table is $d_i \sqrt[n]{k/d}$.*

4.2 Advantage of Replication for Star Joins

Since the shared attributes of a star join are keys of the dimension tables, we do not expect a large blow-up in the size of the join. However, it is normal for the fact table to be orders of magnitude larger than the dimension tables, so there is a definite advantage of not having to communicate the intermediate joins, where the fact table is joined with each dimension table, in turn. Even if the dimension tables are significantly replicated, the cost of communicating the dimension tables can still be much smaller than the cost of communicating the fact table.

There is another question that the result in Section 4.1 answers. Aster Data (www.asterdata.com), lays out fact and dimension tables across a large number of nodes, by partitioning the fact table across the nodes and replicating the dimension tables so that each tuple of each dimension table has a copy at any node with one or more fact-table tuples that joined with it. They viewed the problem as finding an optimum partition of the fact table, taking into account the particular values in the data. The minimum-communication solution we developed in Section 4.1 tells the most space-efficient way to partition the fact and dimension tables, but in a way that is oblivious to the data. We believe that our solution will be the best in practice for two reasons:

1. It is unlikely that typical data distributions allows less replication than the data-oblivious approach.

2. But more importantly, if we take the data into account, then as the fact table grows, we need to rethink the distribution of the dimension tables with each additional tuple. With the data-oblivious approach, we would only add the new fact tuple to the one node to which that tuple hashed.

4.3 Chain Joins

A *chain join* is a join of the form

$$R_1(A_0, A_1) \bowtie R_2(A_1, A_2) \bowtie \cdots \bowtie R_n(A_{n-1}, A_n)$$

as suggested by Fig. 4. It is probably the most common form of join, at least if one includes cases where the relations have attributes other than the A's that are unique to those relations (and whose presence would not affect the analysis we are about to offer).

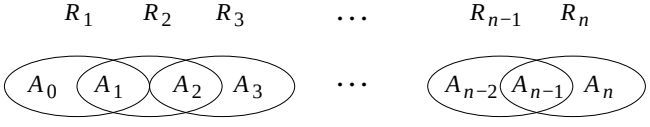


Figure 4: General form of a chain join

EXAMPLE 4.3. *We shall use, as a running example, the specific chain join*

$$R(A, B) \bowtie S(B, C) \bowtie T(C, D) \bowtie U(D, E)$$

In terms of the general chain-join form, $n = 4$, R, S, T, and U play the roles of R_1, R_2, R_3, and R_4, respectively, and the roles of A_0 through A_4 are played by A, B, C, D, and E, respectively.

Let us apply the algorithm of Section 3.5 to a chain join. First, Step 1 tells us to eliminate the attributes A_0 and A_n from the map-key, as they are dominated by A_1 and A_{n-1}, respectively. No other attributes are dominated, so the map key consists of $\{A_1, A_2, \ldots, A_{n-1}\}$.

EXAMPLE 4.4. *For the case of Example 4.3, we eliminate A and E. The map-key consists of B, C, and D.*

In Step 2, we construct the cost expression. It is the sum of terms, one for each relation. The term τ_i for R_i consists of factor r_i and all the share variables a_j where A_j is not in the schema of R_i but is in the map-key. That is, we require either $1 \leq j \leq i-2$ or $i < j < n$.

EXAMPLE 4.5. *Let us continue Example 4.4. The cost expression is*

$$rcd + sd + tb + ubc$$

Note the general pattern. The term corresponding to R_1 will have factors r_1 and all share variables a_j for $2 \leq j < n$. The first term above is an example. The term for R_n will have factors r_n and all share variables a_j where $1 \leq j \leq n-2$; the last term above illustrates. All other terms are like sd and tb above, they have one fewer factor than the end terms, and are missing the share variables for the two attributes of their schema. Note also that as the chain gets longer, the terms get larger. For arbitrary n, the end terms have $n-2$ share variables as factors and the middle terms have $n-3$ share variables as factors.

In what follows, we shall use τ_i to stand for the term constructed in Step 2 for the relation R_i. Then, in Step 3, we construct the Lagrangean equations:

$$\begin{aligned}
&\tau_3 + \tau_4 + \cdots + \tau_n = \lambda k \\
&\tau_1 + \tau_4 + \cdots + \tau_n = \lambda k \\
&\tau_1 + \tau_2 + \tau_5 + \cdots + \tau_n = \lambda k \\
&\tau_1 + \tau_2 + \tau_3 + \tau_6 + \cdots + \tau_n = \lambda k \\
&\quad \cdots
\end{aligned}$$

That is, each equation is missing two consecutive τ's.

If we subtract the first equation from the second, we get $\tau_1 = \tau_3$. Subtracting the second from the third yields $\tau_2 = \tau_4$, and in a similar manner we can derive $\tau_i = \tau_{i+2}$ for all i from $i = 1$ to $i = n-2$. That is, all the even terms are equal, and all the odd terms are equal.

EXAMPLE 4.6. *Following Example 4.5, we get $rcd = tb$ and $sd = ubc$. These equations, together with $bcd = k$ are all we need to get a solution. First, from $rcd = tb$ we get $b = (r/t)cd$. Substitute for b in $sd = ubc$ to get $sd = (ur/t)c^2 d$. The latter equation simplifies to $c = \sqrt{st/ru}$. Notice that c has a value that doesn't depend on k. If $st < ru$, then $c = 1$ must be chosen; i.e., attribute C is not really part of the map-key. However, if $st > ru$, then c has a constant value greater than 1. For example, if $st = 4ru$, then C's share is exactly 2; i.e., we must partition C-values into two buckets.*

We can continue to solve for b and d. From $b = (r/t)cd$ and $c = \sqrt{st/ru}$ we deduce $b = d\sqrt{rs/tu}$ by substituting

for c in the formula for b. Then, since $bcd = k$, we may substitute for b and c to get $d^2\sqrt{st/ru}\sqrt{rs/tu} = k$. From this equation, we solve for d to get $d = \sqrt{ku/s}$. From there, with $b = d\sqrt{rs/tu}$, we get $b = \sqrt{kr/t}$.

4.4 Solving the General Case of Chain Joins

Our goal is to give a closed-form expression for the share belonging to every attribute in a chain join. Interestingly, the solutions are rather different for odd- and even-length chains.

We shall first analyze the case where all relations are of equal size. That case involves considerably simpler algebraic expressions, yet illustrates the two different forms of solution, one for even n and one for odd n. It also serves to introduce the algorithm used in the general case, without obscuring the idea behind the algebra. Figure 5 suggests how the shares of the attributes of the map-key vary along the chain in the two cases.

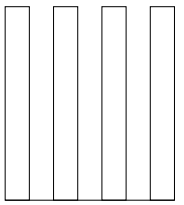
Recall that the Lagrangean equations

$$\begin{aligned}
&\tau_3 + \tau_4 + \cdots + \tau_n = \lambda k\\
&\tau_1 + \tau_4 + \cdots + \tau_n = \lambda k\\
&\tau_1 + \tau_2 + \tau_5 + \cdots + \tau_n = \lambda k\\
&\tau_1 + \tau_2 + \tau_3 + \tau_6 + \cdots + \tau_n = \lambda k\\
&\quad \cdots
\end{aligned}$$

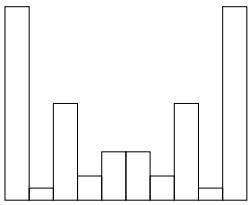
imply that

$$\begin{aligned}
\tau_1 = \tau_3 = \tau_5 = \cdots\\
\tau_2 = \tau_4 = \tau_6 = \cdots
\end{aligned}$$

Moreover, the converse holds as well, in the sense that the equalities among the τ's imply the original equations, with the exception of the fact that we lose the particular value λk. However, since we need to solve for λ anyway, the loss is not important, and we shall henceforth look only for values of the share variables that satisfy the equalities of the even τ's and the odd τ's.



(a) Pattern for even numbers of relations



(b) Pattern for odd numbers of relations

Figure 5: For chains of even length, only alternating attributes get a share; for odd lengths, the shares form an increasing and decreasing sequence, interlaced

We can write τ_1 as r_1k/a_1, τ_n as r_nk/a_{n-1} and all other τ's as $\tau_i = r_ik/(a_{i-1}a_i)$. The fact that k is the product of the a_i's justifies this rewriting. That is, the equalities of the τ's, with common factor k removed, can be written in two simple ways, depending on whether n is odd or even. For even n:

$$\frac{r_1}{a_1} = \frac{r_3}{a_2a_3} = \frac{r_5}{a_4a_5} = \cdots = \frac{r_{n-1}}{a_{n-2}a_{n-1}}$$

$$\frac{r_2}{a_1a_2} = \frac{r_4}{a_3a_4} = \cdots = \frac{r_{n-2}}{a_{n-3}a_{n-2}} = \frac{r_n}{a_{n-1}}$$

Note that for even n, the two end terms, for R_1 and R_n, have their contributions in different equations. These terms differ from all others, in that they have only one a in the denominator. For odd n, one equation has both of the "end" terms, and the other has none:

$$\frac{r_1}{a_1} = \frac{r_3}{a_2a_3} = \frac{r_5}{a_4a_5} = \cdots = \frac{r_{n-2}}{a_{n-3}a_{n-2}} = \frac{r_n}{a_{n-1}}$$

$$\frac{r_2}{a_1a_2} = \frac{r_4}{a_3a_4} = \cdots = \frac{r_{n-1}}{a_{n-2}a_{n-1}}$$

4.4.1 Even n, All Relations Equal in Size

We can simplify the equations by setting $r_1 = r_2 = \cdots = r_n$ and dividing through by the relation size. Further, we can invert each term, so we need to solve the following:

$$\begin{aligned}
&a_1 = a_2a_3 = a_4a_5 = \cdots = a_{n-2}a_{n-1}\\
&a_1a_2 = a_3a_4 = \cdots = a_{n-3}a_{n-2} = a_{n-1}
\end{aligned}$$

It turns out that expressing everything in terms of a_2 is the most effective way to resolve the equations. Remember that we cannot solve for exact values of the a_i's until we apply the condition that their product is k, but we can solve for their ratios. To begin, we can prove by induction on i that:

LEMMA 4.7. *$a_{2i} = (a_2)^i$, for $i = 1, 2, \ldots, (n/2) - 1$.*

Next, use the equality of the ends of both sequences of equal terms to get $a_1 = a_{n-2}a_{n-1}$ and $a_1a_2 = a_{n-1}$. From these, it follows that $a_2a_{n-2} = 1$. But we also have derived $a_{n-2} = (a_2)^{(n/2)-1}$. That is, $a_2(a_2)^{(n/2)-1} = 1$. But all the a_i's are positive numbers, so it follows that a_2 must be 1. Therefore, all the even-subscripted a's are 1; that is

$$a_2 = a_4 = a_6 = \cdots = a_{n-2} = 1$$

Once we know the even a's are all 1, it follows from either set of equalities that the odd a's are all the same; that is: $a_1 = a_3 = a_5 = \cdots = a_{n-1}$. Further, we can use the fact that the product of all the a's is k to deduce that

$$a_1 = a_3 = a_5 = \cdots = a_{n-1} = k^{2/n}$$

That is, all the odd a's are the $(n/2)$th root of k. This solution makes intuitive sense; it says that each of the relations in the join has one of its attributes contributing to the map-key and the other not.

4.4.2 Odd n, All Relations Equal in Size

We have to solve almost the same set of equations, but the two terms a_1 and a_{n-1} that are different because they are not the product of two a's appear in the same set of equalities as:

$$\begin{aligned}
&a_1 = a_2a_3 = a_4a_5 = \cdots = a_{n-3}a_{n-2} = a_{n-1}\\
&a_1a_2 = a_3a_4 = \cdots = a_{n-2}a_{n-1}
\end{aligned}$$

The same argument as in Lemma 4.7 tells us that each of the even-subscripted a's is a power of a_2; that is $a_{2i} = (a_2)^i$. But now, a_{n-1} is one of these, and we deduce $a_{n-1} = (a_2)^{(n-1)/2}$.

We also know from the first set of equalities that $a_1 = a_{n-1}$, so $a_1 = (a_2)^{(n-1)/2}$. Now, we can solve for the remaining odd-subscripted a's. Using the first set of equalities, we know that $a_1 = a_{2i}a_{2i+1}$ for $i = 1, 2, \ldots, (n-3)/2$. We've also deduced that $a_{2i} = (a_2)^i$. Thus, $a_{2i+1} = (a_2)^{((n-1)/2)-i}$. That is, the even-subscripted a's are increasing powers of a_2, while the odd-subscripted a's are decreasing powers of a_2.

Thus, the product of the a's is $(a_2)^{(n-1)(n+1)/4}$. Since this product is k, we find $a_2 = k^{4/((n-1)(n+1))}$, from which we can deduce each of the a's.

EXAMPLE 4.8. *Suppose* $n = 7$ *and* $k = 4096 = 2^{12}$. *Then* $a_2 = 4096^{1/12} = 2$. *It follows that* $a_1 = a_6 = 8$, $a_2 = a_5 = 2$, *and* $a_3 = a_4 = 4$.

In the complete version, we also give the analysis for the case the relations have arbitrary sizes.

5. RELATED WORK

Optimization techniques for the evaluation of join queries in parallel environments has been studied in various settings. Several (e.g., see [11] and references therein) consider piplelined parallelism, which uses a limited number of processors since the parallelism involves assigning a different processor to each operator in the query. Adaptive techniques have been proposed to overcome the bottleneck of one processor being overworked, while the other processors are idle for a long time [3]. Adaptive techniques for queries over stream data are considered in [4,18] and for data integration in [15]. In [21], query optimization is investigated in a similar setting, where, however, each Web service is viewed as a different processor. With a different goal — maximizing the output rate of join queries over streaming data to address the problem of computing a prefix of the query output as soon as possible — the problem is investigated in [23]. The problem of partitioning efficiently the data among a given (possibly large) number of processors has been addressed in [12, 20].

Several works have noticed and investigated issues that concern the computation of an n-way join as one multi-tuple operation [5, 22, 23]. The multiway join operator has been considered in [23] for processing streaming data. Tree-based techniques for executing a multiway join operator for both select-project-join and recursive queries are developed in [16].

Acknowledgment We would like to thank Raghotham Murthy, Chris Olston, and Jennifer Widom for their advice and suggestions in connection with this work. Also we thank Victor Kyritsis for running the experiments.

6. REFERENCES

[1] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. Technical report, Stanford, http://ilpubs.stanford.edu:8090/952/, 2009.

[2] Apache. Hadoop. http://hadoop.apache.org/, 2006.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272, 2000.

[4] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, 2005.

[5] S. Babu and J. Widom. Streamon: an adaptive engine for stream query processing. In *SIGMOD Conference*, pages 931–932, New York, NY, USA, 2004. ACM.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[8] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD Conference*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[11] A. Deshpande and L. Hellerstein. Flow algorithms for parallel query optimization. In *ICDE*, pages 754–763, 2008.

[12] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–40, 1992.

[13] D. J. DeWitt, E. Paulson, E. Robinson, J. F. Naughton, J. Royalty, S. Shankar, and A. Krioukov. Clustera: an integrated computation and data management system. *PVLDB*, 1(1):28–41, 2008.

[14] S. Ghemawat, H. Gobioff, , and S.-T. Leung. The google file system. In *19th ACM Symposium on Operating Systems Principles*, 2003.

[15] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *SIGMOD Conference*, pages 299–310, 1999.

[16] H. Jacobsson. Tree-based techniques for query evaluation. Ph.D. thesis, Dept. of CS, Stanford Univ., Stanford CA USA, STAN-CS-93-1492, 1993.

[17] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46:668–677, 1999.

[18] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, pages 49–60, 2002.

[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[20] K. A. Ross and J. Cieslewicz. Optimal splitters for database partitioning with size bounds. In *ICDT*, pages 98–110, New York, NY, USA, 2009. ACM.

[21] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB*, pages 355–366, 2006.

[22] K.-L. Tan and H. Lu. A note on the strategy space of multiway join query optimization problem in parallel systems. *SIGMOD Rec.*, 20(4):81–82, 1991.

[23] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.