Handling Data Skew in Parallel Joins in Shared-Nothing Systems

Yu Xu
Teradata
San Diego, CA, USA
yu.xu@teradata.com

Pekka Kostamaa
Teradata
El Segundo, CA, USA
pekka.kostamaa@teradata.com

Xin Zhou
Teradata
El Segundo, CA, USA
xin.zhou@teradata.com

Liang Chen
University of California, San Diego
San Diego, CA, USA
jeffchen@cs.ucsd.edu

ABSTRACT

Parallel processing continues to be important in large data warehouses. The processing requirements continue to expand in multiple dimensions. These include greater volumes, increasing number of concurrent users, more complex queries, and more applications which define complex logical, semantic, and physical data models. Shared nothing parallel database management systems [16] can scale up “horizontally” by adding more nodes. Most parallel algorithms, however, do not take into account data skew. Data skew occurs naturally in many applications. A query processing skewed data not only slows down its response time, but generates hot nodes, which become a bottleneck throttling the overall system performance. Motivated by real business problems, we propose a new join geography called PRPD (Partial Redistribution & Partial Duplication) to improve the performance and scalability of parallel joins in the presence of data skew in a shared-nothing system. Our experimental results show that PRPD significantly speeds up query elapsed time in the presence of data skew. Our experience shows that eliminating system bottlenecks caused by data skew improves the throughput of the whole system which is important in parallel data warehouses that often run high concurrency workloads.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: DATABASE MANAGEMENT—*Systems*

General Terms

Algorithms

Keywords

data skew, parallel joins, shared nothing

1. INTRODUCTION

Parallel processing continues to be important in large data warehouses. The processing requirements continue to expand in multiple dimensions. These include greater volume, increasing number of concurrent users, more complex queries, and more applications which define complex logical, semantic, and physical data models.

In a shared nothing architecture, multiple nodes communicate via high-speed interconnect network and each node has its own private memory and disk(s). In current systems, there are usually multiple virtual processors (collections of software processes) running on each node to take advantage of the multiple CPUs and disks available on each node for further parallelism. These virtual processors, responsible for doing the scans, joins, locking, transaction management, and other data management work, are called *Parallel Units (PUs)* in this paper.

Relations are usually horizontally partitioned across all PUs which allows the system to exploit the I/O bandwidth of multiple disks by reading and writing them in parallel. Hash partitioning is commonly used to partition relations across all PUs. Tuples of a relation are assigned to a PU by applying a hash function to their *Partitioning Column*. This Partitioning Column is one or more attributes from the relation, specified by the user or automatically chosen by the system.

As an example, Figure 1 shows the partitioning of two relations $R(x, a)$ and $S(y, b)$ on a three-PU system, assuming that the partitioning columns are $R.x$ and $S.y$ respectively, and that the hash function h is $h(i) = i \ mod \ 3 + 1$. The hash function h places any tuple with the value i in the partitioning column on the $h(i)$-th PU. For example, a tuple $(x = 3, a = 1)$ of R is placed on the first PU since $h(3) = 1$. The fragment of R (or S) on the i-th PU is denoted as R^i (or S^i).

Conventionally, in a shared nothing parallel system, there are two join geographies to evaluate $R \overset{R.a=S.b}{\bowtie} S$. The first join geography is called the *redistribution plan* and the second is called the *duplication plan*. Both plans consist of two stages.

In the first stage of the redistribution plan, when neither $R.a$ or $S.b$ is the partitioning column, both R and S are redistributed based on the hash values of their join attributes so that matching rows are sent to the same PUs[1]. This redistribution in the first stage of the redistribution plan is called *hash redistribution*. For example, Figure 2 shows the result of the first stage after hash redistributing both R and S on $R.a$ and $S.b$ respectively. R^i_{redis} (or S^i_{redis}) denotes the spool on the i-th PU that contains all rows of R (or S) hash redistributed to the i-th PU from all PUs. These include rows from R^i (S^i). Obviously, if a relation's join attribute is its partitioning column, there is no need to hash redistribute that relation. When both relations' join attributes are their partitioning columns, the first stage of the redistribution is not needed.

In the first stage of the duplication plan, tuples of the smaller relation on each PU is duplicated (broadcast) to all PUs so that each PU has a complete copy of the smaller relation. As an example, Figure 3 shows the result of duplicating S in Figure 1 to every PU.

In the second stage of both plans, the join operation is performed on each PU in parallel. This can be done since the first stage has put all matching rows from the join relations on the same PUs.

Research has shown that the redistribution plan has near-linear speed-up on shared nothing systems under even balancing conditions [6]. However, if we use the redistribution plan to evaluate $R \overset{R.a=S.b}{\bowtie} S$ and if one relation (R) has many rows with the same value v in the join attribute ($R.a$), one PU will receive all these rows. This PU will become *hot* and can be the performance bottleneck in the whole system. The value v is called a *skewed value* of R in $R.a$. Any row of R containing a skewed value v is called a *skewed row*. Figure 2 shows that the number of rows of R the second PU receives is 4 times that of any other PU. Obviously the more skew in the data, the hotter the hot PU will become.

Adding more nodes to the system will not solve the skew problem because all skewed rows will still be sent to a single PU. This will only reduce the parallel efficiency, since adding more nodes will make each non-hot PU colder (having fewer rows) and make the hot PU comparatively even hotter. The type of data skew demonstrated in Figure 2 is categorized as *redistribution skew* in [17] and is what we study in this paper.

Redistribution skew could be the result of a poorly designed hash function. However, theoretical research on hashing offers a class of good universal hash functions [4] that perform well with a high probability. The more fundamental problem comes from naturally occurring skewed values in the join attributes.

We have seen redistribution skew in many types of industrial applications. For example, in the travel booking industry, a big customer often makes a large number of reservations on behalf of all of its end users. In online e-commerce, a few professionals make millions of transactions a year while the vast majority of the other customers only do a few a year. In telecommunication, some phone numbers used for telemarketing make a huge number of phone calls while most customers make a few calls daily. In retail, some items sell far more than other items. Usually the relations in these

[1]The base relations are not changed, only copies of the projected rows are redistributed for the evaluation of this query.

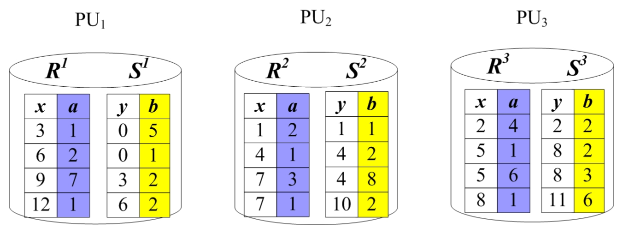


Figure 1: Two relations R and S on 3 Parallel Units. The partitioning columns are $R.x$ and $S.y$ respectively. The hash function, $h(i) = i \ mod \ 3 + 1$, places a tuple with value i in the partitioning column on the $h(i)$-th PU.

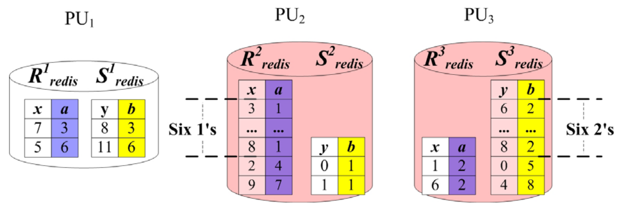
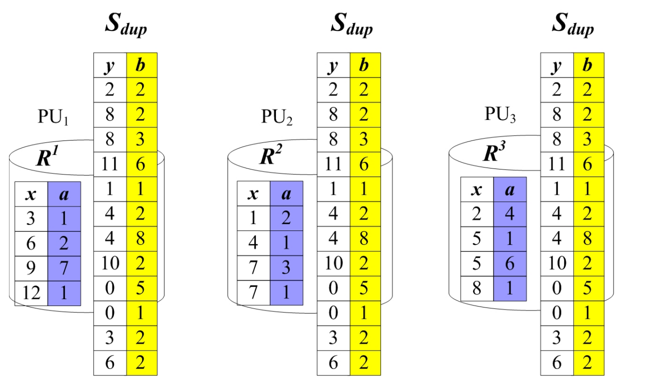


Figure 2: The first stage of the redistribution plan for $R \overset{a=b}{\bowtie} S$: data placement after hash redistributing R and S on the join attributes ($R.a$ and $S.b$) based on the hash function $h(i) = i \ mod \ 3 + 1$. R^i_{redis} (or S^i_{redis}) denotes the spool on the i-th PU that contains all rows of R (or S) hash redistributed to the i-th PU.

applications are almost evenly partitioned by their unique transaction ids. However, when the join attribute is a non-partitioning column attribute (like customer id, phone number or item id), severe redistribution skew happens. This can have a disastrous effect on system performance. Out of spool space errors, which cause queries to abort (often after hours of operation in large data warehouses), can also happen in the presence of severe redistribution skew. This is because although disk continues to become larger and cheaper, parallel DBMSes still maintain spool space quotas for users on each PU for the purpose of workload management and concurrency control. Sometimes, a system can choose the duplication plan instead of the redistribution plan to alleviate the skew problem. This works in very limited cases when one join relation is fairly small. Though many algorithms have been proposed in the research community, to our best knowledge, no effective skew handling mechanism has been implemented by major parallel DBMS vendors, either because of their high implementation complexity or communication cost, or the significant changes required to the current systems.

We make the following contributions in the paper:

- We propose a practical and efficient join geography called Partial Redistribution & Partial Duplication (PRPD) to handle data skew in parallel joins motivated by real business problems.
- The PRPD join geography does not require major changes to the current implementations of a shared-nothing architecture.



PU$_1$ S_{dup}:

y	b
2	2
8	2
8	3
11	6
1	1
4	2
4	8
10	2
0	5
0	1
3	2
6	2

PU$_1$ R^1:

x	a
3	1
6	2
9	7
12	1

PU$_2$ S_{dup}:

y	b
2	2
8	2
8	3
11	6
1	1
4	2
4	8
10	2
0	5
0	1
3	2
6	2

PU$_2$ R^2:

x	a
1	2
4	1
7	3
7	1

PU$_3$ S_{dup}:

y	b
2	2
8	2
8	3
11	6
1	1
4	2
4	8
10	2
0	5
0	1
3	2
6	2

PU$_3$ R^3:

x	a
2	4
5	1
5	6
8	1

Figure 3: The first stage of the duplication plan for $R \overset{a=b}{\bowtie} S$: data placement after duplicating S to every PU. The spool S_{dup} on each PU has a complete copy of S.

- Our experiments show the efficiency of the proposed PRPD join geography.

The rest of the paper is organized as follows. In Section 2 we present the PRPD join geography. Section 3 discusses how the PRPD geography is applied to multiple joins. Section 4 shows the experimental results. Section 5 discusses related work. Section 6 concludes the paper and discusses future work.

2. PARTIAL REDISTRIBUTION PARTIAL DUPLICATION (PRPD)

In this section we present the PRPD join geography, starting with an observation and an intuition behind the PRPD approach. We notice that parallel system DBAs and application writers are well educated on the importance of choosing good partitioning columns to evenly partition their data for efficient parallel processing, usually through required certification or training. Parallel DBMSes provide tools to show how data is partitioned and, if necessary, to help users change the partitioning columns. In the absence of other good choices, tables can be defined using identity columns (which automatically generate unique values) as partitioning columns. Thus, in practice, large relations are most likely to be evenly partitioned.

Consider the join $R \overset{R.a=S.b}{\bowtie} S$. Assume R is evenly partitioned, $R.a$ is not the partitioning column of R, and R has many skewed rows in $R.a$. The key observation is that the skewed rows of R tend to be evenly partitioned on each PU. The intuition behind the PRPD join geography is to deal with the skewed rows and non-skewed rows of R differently. PRPD keeps the skewed rows of R locally on each PU (instead of hash redistributing them all to a single PU), and duplicates those matching rows from S to all PUs. PRPD uses the redistribution plan for other rows of R and S. In Section 2.1 we discuss the PRPD join geography. Section 2.2 compares PRPD with the redistribution plan and the duplication plan. Section 2.3 discusses how PRPD handles unusual cases where skewed rows are not evenly partitioned on all PUs.

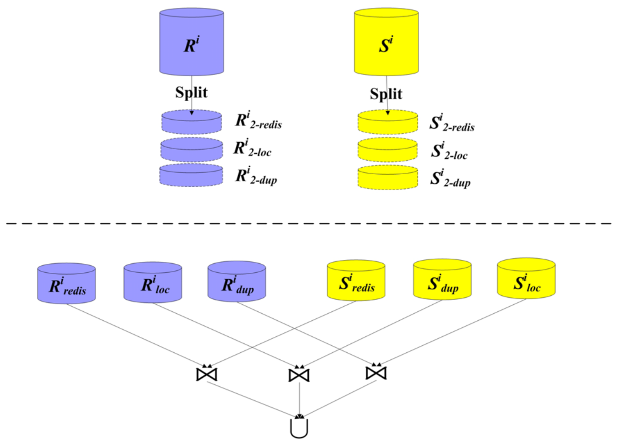


Figure 4: the PRPD plan for $R \overset{a=b}{\bowtie} S$: $R^i_{2-redis}$ and $S^i_{2-redis}$ are hash redistributed on $R.a$ and $S.b$ respectively; R^i_{2-dup} and S^i_{2-dup} are duplicated; R^i_{2-loc} and S^i_{2-loc} are kept locally.

2.1 PRPD description

Assume the optimizer chooses the PRPD join geography to join $R \overset{R.a=S.b}{\bowtie} S$. For simplicity, we will use the terms PRPD join geography and the PRPD plan interchangeably. We first assume that the system knows the set of skewed values L_1 in $R.a$ and the set of skewed values L_2 in $S.b$ and that neither $R.a$ nor $S.b$ is the partitioning column. We will show later how PRPD works when only one relation is skewed or when one relation's partitioning column is the join attribute.

There are three steps in the PRPD plan. Assume there are n PUs in the system.

- Step 1
  - 1) On each PU_i, scan R^i once and split the rows into three sets. The first set, named R^i_{2-loc}, contains all skewed rows of R^i for any value in L_1. R^i_{2-loc} is kept locally. The second set, named R^i_{2-dup}, contains every row of R^i whose $R.a$ value matches any value in L_2 and is duplicated to all PUs. The third set, named $R^i_{2-redis}$ contains all other rows in R^i and is hash redistributed on $R.a$. Note that R^i_{2-loc}, R^i_{2-dup} and $R^i_{2-redis}$ disjointly partition R^i on each PU.

    On each PU_i, three receiving spool files R^i_{redis}, R^i_{dup}, and R^i_{loc} are created. The first spool R^i_{redis} receives all rows of R redistributed to the i-th PU from any PU including PU_i itself. That is, R^i_{redis} contains any row from any $R^j_{2-redis}$ $(1 \leq j \leq n)$ which is redistributed to the i-th PU by the system's hash function, h.

    $$R^i_{redis} = \{\tau \mid h(\tau.a) = i \,\&\, \tau \in \bigcup_{1 \leq j \leq n} R^j_{2-redis}\}$$

    The second spool R^i_{dup} receives all rows of R duplicated to the i-th PU from any PU including PU_i itself. That is, $R^i_{dup} = \bigcup_{1 \leq j \leq n} R^j_{2-dup}$. Notice that $R^i_{dup} = R^j_{dup} = \bigcup_{1 \leq p \leq n} R^p_{2-dup}$ for any

i and j. The third spool R^i_{loc} receives all rows from R^i_{2-loc}. Mathematically, $R^i_{loc} = R^i_{2-loc}$.

- 2) Similarly, on each PU_i, scan S^i once and split the rows into three sets. The first set, named S^i_{2-loc}, contains all skewed rows of S^i in any value in L_2. S^i_{2-loc} is kept locally. The second set, named S^i_{2-dup}, contains every row of S^i whose $S.b$ value matches any value in L_1 and is duplicated to all PUs. The third set, named $S^i_{2-redis}$ contains all other rows in S^i and is hash redistributed on $S.b$. Note that S^i_{2-loc}, S^i_{2-dup} and $S^i_{2-redis}$ disjointly partition S^i on each PU.

On each PU_i, three receiving spools S^i_{redis}, S^i_{dup}, and S^i_{loc} are also created, similarly to R^i_{redis}, R^i_{dup}, and R^i_{loc}. The definitions are shown below.

$$S^i_{redis} = \{\tau \mid h(\tau.b) = i \ \& \ \tau \in \bigcup_{1 \le j \le n} S^j_{2-redis}\}$$

$S^i_{dup} = \bigcup_{1 \le j \le n} S^j_{2-dup}$
$S^i_{loc} = S^i_{2-loc}$
Note that in implementation, when a row of R or S is read, the system immediately determines whether to keep it locally, redistribute it or duplicate it based on the value in the join attribute. The six sets $R^i_{2-redis}$, R^i_{2-dup}, R^i_{2-loc}, $S^i_{2-redis}$, S^i_{2-dup} and S^i_{2-loc} are not materialized and are used only for presentation purpose. Only the six receiving spools R^i_{redis}, R^i_{dup}, R^i_{loc}, S^i_{redis}, S^i_{dup} and S^i_{loc} are materialized and used in the join operations shown in the next step.

- Step 2 On each PU_i do the following three joins,
  - 1) $R^i_{redis} \overset{a=b}{\bowtie} S^i_{redis}$.
  - 2) $R^i_{loc} \overset{a=b}{\bowtie} S^i_{dup}$.
  - 3) $R^i_{dup} \overset{a=b}{\bowtie} S^i_{loc}$.
- Step 3
  The final result is the union of the three joins from Step 2 on all PUs.

Notice that all sub-steps in each step can run in parallel and they usually do. For example, the sub-steps 1.1 and 1.2 in the first step can run in parallel, and so can the three joins in the second step.

In Step 1, the skewed values in $R.a$ and $S.b$ together determine how to split R and S into different spools. When a value v appears in both the skewed values in $R.a$ and $S.b$, we can not duplicate them in both relations. Our solution dealing with overlapping skewed values is described below. When invoking the PRPD plan, the system always invokes the PRPD join plan with two non-overlapping sets L_1 (of skewed values in R) and L_2 (of skewed values in S). The systems chooses to include the skewed value v (in both relations) in only one of L_1 and L_2, instead of in both. By doing this, it is guaranteed that R^i_{2-loc}, R^i_{2-dup} and $R^i_{2-redis}$ disjointly partition R^i on each PU, and S^i_{2-loc}, S^i_{2-dup} and $S^i_{2-redis}$ disjointly partition S^i on each PU. The question is to decide which set (L_1 or L_2) should include v. Assume the number of skewed rows of R (whose $R.a$ value is v) times the projected row size of R is bigger than the number of skewed rows of S (whose $S.b$ value is v) times the projected row size of S, then the system chooses to include v in L_1 to keep these skewed rows of R locally in R^i_{loc}, and those rows of S whose join attribute value is v are duplicated to S^i_{dup}.

When only one relation is skewed, the first step produces two receiving spools for each relation instead of three. Assume only R is skewed. The first step will produce only two receiving spools R^i_{loc} and R^i_{redis} for R, and S^i_{dup} and S^i_{redis} for S. R^i_{dup} and S^i_{loc} are not created since there is no skewed value in S. Consequently, the second step will only need to perform the first two joins. In Step 2, we allow the optimizer to choose the best physical join method for each of the three joins. The optimizer may choose different join methods for each of the three joins.

When one relation's (assume it is S) join attribute is also its partitioning column, the PRPD approach works logically in the same way. The one difference is that each PU will keep locally those rows of S whose join attribute values are not skewed in R, rather than redistribute them to any other PUs since $S.b$ is the partitioning column. Although the redistribution plan does not need to redistribute S, skewed rows in R will be still sent to a single PU, causing system bottleneck. In this case, PRPD can outperform the redistribution plan just as in the case where neither relation's join attribute is the partitioning column.

Overall, when R is skewed, the skewed rows of R are kept locally, the matching rows in S for these skewed rows are duplicated to all PUs, and all other rows from both relations are redistributed. Part of S is redistributed and part of S is duplicated. When S is also skewed, symmetric actions are taken. Thus we named the proposed approach PRPD (partial redistribution and partial duplication).

The basic idea of the PRPD plan is illustrated in Figure 4. The new operator *Split* reads each row and either keeps it locally, redistributes it or duplicates it according to its value in the join attribute and the set of skewed values in R and S. As an example, Figure 5(a) shows how the PRPD plan is going to place each row of R and S at each PU for the example data shown in Figure 1. Figure 5(b) shows the data placement after applying the PRPD plan. Comparing Figure 5(b) with Figure 2, we can see the figures visually show that the processing at each PU in the PRPD plan is even and there is no hot PU.

Appendix A provides the correctness proof of the PRPD plan.

2.2 PRPD, redistribution plan and duplication plan comparison

The Partial Duplication part of the PRPD plan is not in the redistribution plan, therefore causes PRPD to use more total spool space than the redistribution plan. On the other hand, the Partial Redistribution part of the PRPD plan keeps the skewed rows locally and has less networking communication cost than the redistribution plan. A more important difference between the PRPD plan and the redistribution plan is that PRPD does not send all skewed rows to a single PU and does not cause a system bottleneck.

The Partial Duplication part of the PRPD plan may make PRPD use less total spool space than the duplication plan since not every row of a relation is duplicated. On the other

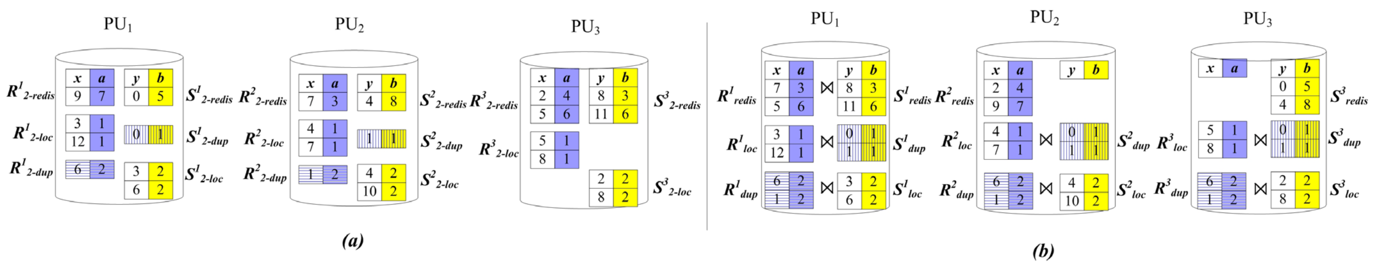


Figure 5: The PRPD join geography for $R \overset{a=b}{\bowtie} S$. Assume the hash function is $h(i) = i \ mod \ 3 + 1$, 1 is the skewed value in R and 2 is the skewed value in S. (a) illustration of each tuple's future placement. $R^i_{2-redis}$ and $S^i_{2-redis}$ are to be hash redistributed on $R.a$ and $S.b$. R^i_{2-dup} and S^i_{2-dup} to be duplicated; R^i_{2-loc} and S^i_{2-loc} to be kept locally. (b) illustration of each tuple's placement after applying the PRPD plan.

hand, the Partial Redistribution part of the PRPD plan may make the PRPD plan have more networking communication cost than the duplication plan when data skew is not significant and the PRPD plan needs to redistribute a large relation. Another difference is that the duplication plan always joins a complete copy of the duplicated relation to the fragment of the other relation on every PU while the PRPD plan reduces the sizes of the spools to be joined on each PU.

The PRPD join plan is a hybrid of the redistribution plan and the duplication plan. Consider the join $R \overset{R.a=S.b}{\bowtie} S$. If the optimizer invokes the PRPD plan with an empty set L_1 (of skewed values in $R.a$) and an empty set L_2 (of skewed values in $S.b$), PRPD will behave exactly the same way as the redistribution plan, since no rows are kept locally or duplicated. If L_1 (the set of skewed values in $R.a$) is equal to $Uniq(R.a)$, the set of unique values in $R.a$, and is a superset of $Uniq(S.b)$, the set of unique values in $S.b$, then PRPD will behave exactly the same way as the duplication plan since every row of R will be kept locally (because every value in $R.a$ is a skewed value) and every row of S will be duplicated. If L_1 is equal to $Uniq(R.a)$, but is not a superset of $Uniq(S.b)$, then PRPD will keep every row of R locally, duplicate some rows of S and hash redistribute some rows of S. In this case those hash redistributed rows of S will not match any rows of R. If L_1 is a superset of $Uniq(S.b)$ and a subset of $Uniq(R.a)$, then PRPD will duplicate every row of S, keep some rows of R locally, and hash redistribute some rows of R. In this case those hash redistributed rows of R will not match any rows of S. Neither hash redistribution in the last two cases is necessary. However unless the optimizer knows in advance the relationships among L_1, $Uniq(R.a)$ and $Uniq(S.b)$, the system has no way to eliminate it.

Assume the number of PUs in the system is n, x is the percentage of the skewed rows (*skewness*) in a relation R. The number of rows of R in the hot PU after redistribution in the redistribution plan is roughly $x|R| + \frac{(1-x)|R|}{n}$ while the number of rows of R on a non-hot PU is roughly $\frac{(1-x)|R|}{n}$. Thus, the ratio of skewed rows of R on the hot PU over the number of rows of R on a non-hot PU is roughly $1 + \frac{nx}{1-x}$. In the PRPD plan, after the first stage, the number of rows of R on each PU is roughly $\frac{x|R|}{n} + \frac{(1-x)|R|}{n} = \frac{|R|}{n}$. Thus, the ratio of the number of rows of R on the hot PU in the redistribution plan over the number of rows of R on any PU in the PPRD plan after the first stage is roughly $1+(n-1)x$.

The ratio depends on the skewness of R and the size of the system (the number of PUs in the system), but does not depend on the size of the relation R itself. The ratio is nearly linear to the size of the system and the skewness of R. The ratio is one important factor determining the speedup ratio that PRPD can achieve over the redistribution plan.

As an example, for a small system where $n = 100$ (4 nodes each running 25 PUs), the ratio is 2 for $x = 1\%$, 3 for $x = 2\%$. For a medium size system where $n = 500$ (20 nodes each running 25 PUs), the ratio is 6 for $x = 1\%$, 11 for $x = 2\%$. For a large system where $n = 5000$ (200 nodes each running 25 PUs), the ratio is 51 for $x = 1\%$ and 103 for $x = 2\%$. We include in Section 4 experiments exploring the impact of the size of the system and data skewness on the speedup ratio of PRPD over the redistribution plan.

One important issue is how the optimizer chooses PRPD over other plans. As usual, a cost based optimizer will choose the PRPD plan over other plans only when it determines the cost of applying PRPD is smaller. We will leave out the discussion on how the optimizer computes and compares the costs of PRPD, redistribution plan and duplication plan, which is beyond the scope of this paper.

In addition to using statistics, the optimizer can also use sampling to find skewed values as sampling is a negligible cost in query response time (with or without data skew) as shown in [7, 8].

2.3 Handling unevenly partitioned skewed rows in PRPD

The PRPD join plan presented so far works on the assumption that skewed rows are roughly evenly partitioned on all PUs. This section discusses how the PRPD plan handles unusual cases where the skewed rows are only on one or a few PUs.

Consider the join $R \overset{R.a=S.b}{\bowtie} S$ and R is skewed in its join attribute. Consider a case where all the skewed rows of R are on a small number of PUs, called *skewed PUs* in this section. Though the PRPD plan will not cause redistribution skew itself since all skewed rows are kept locally, these skewed PUs will become hot PUs since in addition to the skewed rows they already have, they will receive redistributed rows from other PUs. Consider the skewed PU P that contains the most number of skewed rows among all PUs. Assume R is evenly partitioned, the number of PUs is n and the skewness of R is x. With PRPD, P will keep all of its skewed rows and receive some non-skewed rows sent from other PUs. The number of rows on P after the first step in PRPD is roughly

$min(\frac{|R|}{n}, x|R|) + \frac{|R|(1-x)}{n}$ where the function $min(a, b)$ returns the smaller number from a and b. The PRPD plan is slightly modified to handle skewed PUs cases. Instead of keeping the skewed rows of R locally on each PU, we randomly redistribute them to all PUs so that skewed rows are evenly redistributed to all PUs. There is no change on handling S. Those rows of S that contain skewed values in R are still duplicated to every PU. With this change in PRPD, the skewed PU P will now have $\frac{|R|}{n}$ rows of R after the first step in the modified PRPD plan. The ratio of the number of rows on the skewed PU P using the original PRPD plan over the number of rows on the same PU P using the modified PRPD plan is $min(\frac{|R|}{n}, x|R|) + \frac{|R|(1-x)}{n} / \frac{|R|}{n} = min(1, nx) + (1-x)$. The ratio is 1 when $x = 0$ or $x = 1$. When $nx \geq 1$, the ratio is $2-x$. Notice that $\lceil nx \rceil = \lceil \frac{x|R|}{\frac{|R|}{n}} \rceil$ is the minimal number of PUs needed to contain all skewed rows in R , assuming R is evenly partitioned. The ratio shows that redistributing the skewed rows to all PUs in the case where skewed rows are on only one or a few PUs can partition the skewed relation in preparation for the joins in the second stage better than keeping all skewed rows locally.

3. APPLYING PRPD IN MULTIPLE JOINS

In this section we discuss how the PRPD join geography is applied in multiple joins. Intuitively, the fact that not all rows in the join result from applying the PRPD plan are partitioned by the join attributes may be an issue in multiple joins.

In this section we show how the PRPD plan can be mixed with other conventional plans in multi-table join queries. We will not discuss join ordering and planning with the introduction of PRPD, which is beyond the scope of this paper. We focus on how the join result from applying the PRPD plan can be joined with other relations using different join geographies.

Consider a query that joins three relations R, S and Y and the join condition on R and S is $R.a = S.b$. Assume the optimizer chooses to first join R and S. Let X be the result of joining R and S. There are 6 possible cases in terms of join geography that may be used to join X and Y before the introduction of PRPD. For each of X and Y, we can keep it locally, redistribute it or duplicate it. Thus, there are 9 combinations for X and Y. However, when one relation is duplicated, the other relation does not need to be duplicated or redistributed. The 6 possible combinations are shown in Figure 6 where each line denotes one possible case described below.

- Case 1: duplicate X; keep Y locally.
- Case 2: keep X locally; duplicate Y.
- Case 3: redistribute X on its join attribute that is not $R.a$ or $S.b$; redistribute Y.
- Case 4: redistribute X on its join attribute that is not $R.a$ or $S.b$; keep Y locally when Y is a base table and the join attribute of Y is Y's partitioning column or when Y is an intermediate result and has been redistributed on its join attribute.
- Case 5: redistribute Y, keep X locally when the join attribute of X is $R.a$ or $S.b$.
- Case 6: keep both X and Y locally when Y is a base table and the join attribute of Y is Y's partitioning column or when Y is an intermediate result and has been redistributed on its join attribute; and when the join attribute of X is $R.a$ or $S.b$.

Now let us consider the impact of joining R and S using PRPD in the context of multi-table joins. Assume the optimizer has chosen PRPD to join R and S. The skew handling using PRPD has no impact in the first four cases, in the sense that the system can execute the specified join geography in each case as usual. In cases 5 and 6, we cannot simply keep X locally anymore when PRPD has been applied. Certainly we can redistribute X again on R.a or S.b to join X and Y. However we do not need to redistribute every row in X since some rows have been already redistributed to the right PUs in PRPD. We just need to redistribute the results from joining those rows that were kept locally in PRPD.

PRPD is modified to work in multi-table joins as follows. When the optimizer identifies that the join geography of joining X and Y is either case 5 or 6, it uses the following modified PRPD step with the changes highlighted.

Step 2

- 1) Join R^i_{redis} and S^i_{redis}.
- 2) Join R^i_{loc} and S^i_{dup}. **Hash Redistribute the result on R.a**
- 3) Join R^i_{dup} and S^i_{loc}. **Hash Redistribute the result on R.a**

When considering the impact of PRPD in multiple joins, the optimizer takes into account of the cost of the extra redistribution step described above that is necessary for cases 5 and 6.

Note that these six cases are all based on traditional join geographies. PRPD introduces one more choice for the optimizer in joining X and Y in multi-table joins. With PRPD, another choice is to use the PRPD plan again to join X and Y. Indeed, this is a likely choice for the optimizer for cases 5 and 6.

Consider the case where $R.a$ has skewed values and S is not skewed (called "single-skew" in [14], which is also the case most previous work focuses on). If the tuples in R with a skewed value v_1 match some tuples in S and either $R.a$ or $S.b$ is the join attribute in the second join (as in case 5 or case 6), v_1 tends to be a skewed value in X as well, as shown below. Assume the set of unique values that appear in both $R.a$ and $S.b$ is $\{v_1, \ldots, v_k\}$. Let m_i and n_i be the number of tuples containing v_i in $R.a$ (in R) and in $S.b$ (in S) respectively ($1 \leq i \leq k$). Assume v_1 is a skewed value in $R.a$. The number of tuples in the result of joining R and S with the value v_1 in the join attribute is: $m_1 n_1$. The total number of tuples in the join result is: $\sum_{i=1}^{k} m_i n_i$. Thus, the percentage of tuples in the join result containing v_1 in the join attribute is:

$$\delta_2 = \frac{m_1 n_1}{\sum_{i=1}^{k} m_i n_i}$$

Let δ_1 be the skewness of v_1 in R (the percentage of rows in R containing v_1 in $R.a$) and n_{max} be the maximal value in

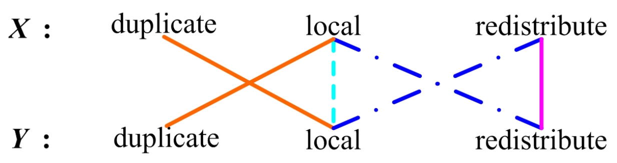


Figure 6: Join geographies for $X \bowtie Y$

$\{n_1, \ldots, n_k\}$. We have

$$\begin{aligned}\delta_2 &= \frac{m_1 n_1}{\sum_{i=1}^{k} m_i n_i} \\ &\geq \frac{m_1 n_1}{\sum_{i=1}^{k} m_i n_{max}} \\ &= \frac{n_1}{n_{max}} \frac{m_1}{\sum_{i=1}^{k} m_i} \\ &\geq \frac{n_1}{n_{max}} \frac{m_1}{|R|} \\ &= \frac{n_1}{n_{max}} \delta_1\end{aligned}$$

The above equation shows that when v_1 is a skewed value in R, v_1 tends to be a skewed value in the join result as well, especially when S is not skewed and n_1 is close to n_{max}.

4. EXPERIMENTAL EVALUATION

In this Section, we are mainly interested in comparing the performances of the PRPD plan and the redistribution plan because the redistribution plan is more widely used in large data warehousing systems than the duplication plan. Since duplicating rows is expensive in terms of both I/O and CPU cost on all PUs and the network bandwidth, the larger the number of PUs in the system and the larger the join relations, the less likely the duplication plan can outperform the redistribution plan or the PRPD plan. However, when one join relation is fairly small, the duplication plan can outperform the other two plans, and we have seen this in our experiments. In this Section, we only report the experiments comparing PRPD and the redistribution plan.

We use the TPC-H [1] and the following query in our experiments.

```
select  *
from    CUSTOMER, SUPPLIER                Q1
where C_NATIONKEY = S_NATIONKEY
```

The schema of the Customer and Supplier relations are shown in Figure 7.

Originally there are only 25 unique nations in the TPC-H database. We have increased the number of unique nations to 1000 to facilitate our skew experiments. To control data skewness in the Customer relation's join attribute (C_NATIONKEY), we randomly choose a portion of the data, and change their C_NATIONKEY values to one value. For example, in our experiments, if the skewness is 5% in the Customer relation's join attribute, this means that we have used the following SQL statement to change 5% of the rows of the Customer relation to have the same value (24) in the join attribute.

```
update CUSTOMER
set    C_NATIONKEY = 24                Q2
where  random(1,100) ≤ 5
```

CUSTOMER (C_)

CUSTKEY
NAME
ADDRESS
NATIONKEY
PHONE
ACCTBAL
MKTSEGMEN
COMMENT

SUPPLIER (S_)

SUPPKEY
NAME
ADDRESS
NATIONKEY
PHONE
ACCTBAL
COMMENT

Figure 7: Two relations *CUSTOMER* and *SUPPLIER* from TPC-H. The partitioning columns are *C_CUSTKEY* and *S_SUPPKEY* respectively.

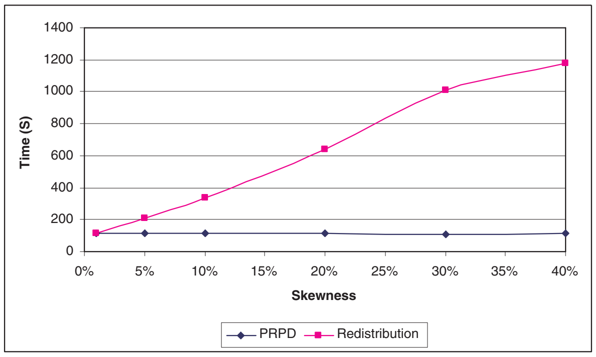


Figure 8: Query execution time, a single skewed value in the Customer relation

4.1 Query execution time

The test system we use for the experiments has 10 nodes and 80 PUs. Each node has four Pentium IV 3.6 GHz CPUs, 4 GB memory and 8 PUs. We generate 1 million rows for the Supplier relation, and 1 million rows for the Customer relation and we vary the data skewness in the Customer relation. The size of the query result is around 1 billion rows. Figure 8 shows the execution times using the PRPD plan and the redistribution plan. When the skewness increases, the execution time of the redistribution plan grows almost linearly because all skewed values are redistributed to one PU and that PU becomes the system bottleneck while the execution time of the PRPD plan essentially stays the same.

We consider the case where there are two skewed values and they cause two hot PUs in the redistribution plan (rows with different skewed values in the join attribute are hash redistributed to two different PUs). We report the results in Figure 9 where both skewed values have the same skewness shown on x-axis. Figures 9 shows the same performance relationship between the two plans as in Figure 8. We also consider the case where both relations have skewed values. The results are similar to what are shown in Figures 8 and 9.

4.2 Speedup ratio with different number of PUs

In this experiment, we calculate the speedup ratio of PRPD over the redistribution plan on three systems, 24 PUs, 56 PUs and 72 PUs respectively. The result is shown in Figure 10. As can been from Figure 10, when the skewness increases, the speedup ratio of PRPD over the redistribu-

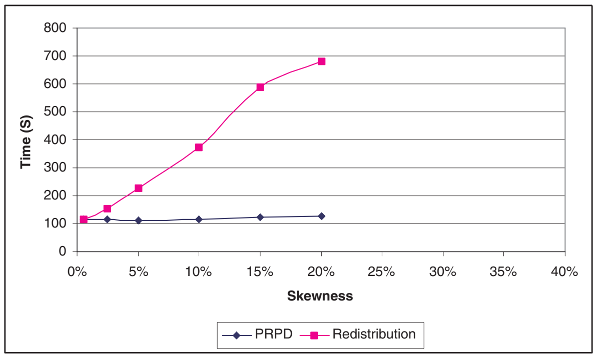


Figure 9: Query execution time, two skewed values with the same skewness shown on x-axis in the Customer relation causing two hot PUs in the redistribution plan

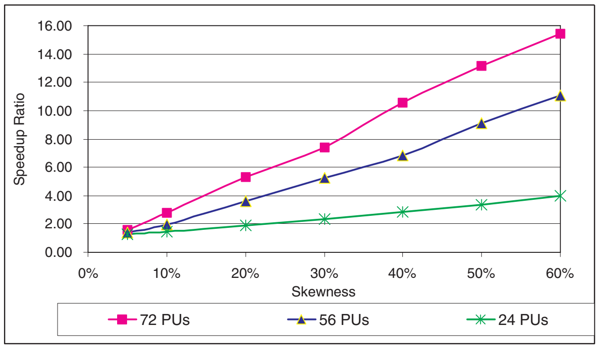


Figure 10: Speedup ratio of the PRPD plan over the redistribution plan

tion plan increases. Also the larger the system, the larger the speedup ratio is.

We have also done experiments on joining two relations where one relation's partitioning column is the join attribute and we have seen similar performance relationships shown in Figures 8, 9 and 10. We note here that our experimental results on using PRPD in multi-table joins also show significant query performance improvement.

5. RELATED WORK

Extensive research has been done on handling data skew in parallel DBMSes. [17] describes four types of skew : tuple placement skew (TPS), selectivity skew (SS), redistribution skew (RS) and join product skew (JPS). TPS can be avoided with a good hash function. In the theoretical literature, hash functions have been extensively studied, and several techniques have been proposed to design a well-performed hash function [4]. Furthermore, in practice partitioning columns are carefully chosen by DBAs so that tuples of large relations are most likely evenly partitioned on all PUs (parallel units). Thus, the possibility of TPS is extremely low and nearly no work focuses on TPS. SS is caused by different selectivity of selection predicates and is query-dependent. SS becomes a problem only when it causes RS or JPS. Most prior work does not consider SS either.

As discussed in Section 1, RS occurs when PUs receive different numbers of tuples when they are redistributed in preparation for the actual joins. JPS occurs when the join selectivity at each node differs, causing imbalance in the number of tuples produced at each PU. RS and JPS are closely related problems and have been heavily studied. Previous algorithms can be roughly classified into the following three categories.

The first category of algorithms models the skew problem as the conventional task scheduling problem. For parallel joins, one or both join relations are partitioned into smaller units (ranges [7] or hash buckets [13, 11, 2, 19, 18, 20, 5]) which are distributed to all PUs. Joins are computed in parallel locally. Since the task scheduling problem is NP-complete, various well-known heuristics including LPT [9] and MULTIFIT [12] are used in this type of algorithms [7, 13, 11, 2, 19, 18, 20]. Based on different cost models, different algorithms in the first category have been proposed.

The first type of models uses the number of tuples on each PU as the estimation of join cost. The goal is to make every PU receive similar number of tuples of the two join relations so that RS does not occur. A few range-partitioning based algorithms in [7] handle the RS problem. The algorithms first find the skewed relation by sampling both relations, and then compute a *split vector* (different algorithms use different strategies to compute the split vector) to split the join attribute values in the skewed relation to a few ranges such that the number of ranges is equal to the number of PUs in the system. Tuples whose join attribute values fall within one range are all *mapped* (redistributed) to the same PU. The goal of the split vector is to make each PU receive similar number of tuples. If one value spans more than one range (which means it is a skewed value), the *subset-replication* algorithm in [7] replicates (duplicates) the tuples in the other join relation with this join attribute value to all PUs that this value maps to. Notice that in the subset-replication algorithm skewed rows are usually redistributed to a few PUs according to the split vector, not to all PUs.

The second type of algorithms uses more sophisticated models ([7, 2, 19, 18, 20, 5]). The *Virtual Processor Scheduling (VPS)* algorithm in [7] deals with the JPS problem. It uses $|R| + |S| + |R \bowtie S|$ (R and S are the two join relations) to estimate the cost of join on each PU. In [2], two functions, *output function* and *work function* are used to estimate join cost. [5] considers parallel systems whose PUs may not have identical configurations.

Unlike the first category of algorithms which try to estimate the execution time before the actual join computation, the core idea of the second category of algorithms is to handle skew dynamically. Work load at each PU is monitored at run time. If a PU is doing much more work than others, some work load from this hot PU will be migrated to other PUs. [15] uses shared virtual memory to migrate workload from hot PUs to idle PUs. [10] detects the processing rate of the join relations on each PU. If a PU's processing rate is slow, this PU is deemed as hot. Two algorithms are introduced: *result redistribution* and *processing task migration*. In *result redistribution*, results generated by hot PUs are not stored locally but sent to other PUs' disk. In *processing task migration*, part of unprocessed relations on a hot PU is migrated to other idle PUs. [21] proposes two algorithms: the single-phase and the two-phase scheduling algorithms. The single-phase algorithm is similar to [7, 13, 11]. In the two-phase scheduling algorithm, a central coordinator is used to maintain a task pool. When a PU becomes idle, it picks up one task from the central pool.

The third category of algorithms uses semi-joins to more accurately estimate the size of joins and to reduce the cost of network communication. [3] proposes two methods. The first method consists of two stages. In the first stage, it computes the histogram of one join relation (R). Based on the histogram, redistribution plan is computed such that each PU receives similar number of R's tuples. In this stage, rows with skewed values are randomly sent to other PUs. In the second stage, the other relation (S) is *semi-join*ed with the histogram of R. The result of the *semi-join* is duplicated to all PUs and joined with R. The second method in [3] is a hybrid approach. It uses the first method only for skewed values and uses other conventional algorithms for non-skewed values.

Compared with previous algorithms, PRPD has the following characteristics. Skewed values are kept locally rather than redistributed to all (or some) PUs. This saves redistribution cost and decreases execution time. Unlike most previous work, PRPD not only deals with cases where only one relation is skewed, but also cases where both relations are skewed. PRPD also solves the type of JPS problems that result from redistribution skew, though it is mainly designed to solve the redistribution skew problem that we have seen in various industrial applications discussed in Section 1. One big advantage of the PRPD algorithm is that it does not require major changes to the implementations of the shared-nothing architecture since it does not require any new type of central coordination or communication among PUs.

6. CONCLUSIONS AND FUTURE WORK

One of the important challenges in parallel DBMSes is to effectively handle data skew in joins. Based on our observations of data warehouses in practice at various industries, we notice that skewed rows tend to be evenly partitioned on all parallel units. Motivated by the redistribution skew problem that arises naturally in business applications in the various industries, we propose an approach called PRPD (partial redistribution & partial duplication). To identify the skewed values, PRPD uses either collected or sampled statistics. Instead of redistributing these skewed rows as is done for the non-skewed rows, they are kept locally. The matching skewed value rows from the joined table are duplicated to each parallel unit to complete the join. The PRPD algorithm also handles the extreme cases where skewed rows are clustered in only one or a few parallel units by redistributing the skewed rows to all parallel units.

Eliminating skewed processing eliminates system bottlenecks created by more conventional algorithms. Our experimental results have demonstrated the effectiveness of the PRPD algorithm in improving query execution time. We have also shown that the PRPD algorithm can be used in multiple joins. In addition, we are looking at handling skew dynamically to avoid reliance on collected or sampled statistics.

7. REFERENCES

[1] TPC Benchmark H (decision support) standard specification http://www.tpc.org.

[2] K. Alsabti and S. Ranka. Skew-insensitive parallel algorithms for relational join. In *HIPC*, page 367, 1998.

[3] M. Bamha and G. Hains. Frequency-adaptive join for shared nothing machines. *Progress in computer research*, pages 227–241, 2001.

[4] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[5] H. M. Dewan, M. A. Hernández, K. W. Mok, and S. J. Stolfo. Predictive dynamic load balancing of parallel hash-joins over heterogeneous processors in the presence of data skew. In *PDIS*, pages 40–49, 1994.

[6] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

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

[8] FrankăOlken and DoronăRotem. Random sampling from databases: a survey. *Statistics and Computing*, 5(1):25–42, 1995.

[9] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[10] L. Harada and M. Kitsuregawa. Dynamic join product skew handling for hash-joins in shared-nothing database systems. In *DASFAA*, pages 246–255, 1995.

[11] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, pages 525–535, 1991.

[12] E. G. C. Jr., M. R. Garey, and D. S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.*, 7(1):1–17, 1978.

[13] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *VLDB*, pages 210–221, 1990.

[14] M. S. Lakshmi and P. S. Yu. Effectiveness of parallel joins. *IEEE Transactions on Knowledge and Data Engineering*, 2(4):410–424, 1990.

[15] A. Shatdal and J. F. Naughton. Using shared virtual memory for parallel join processing. In *SIGMOD Conference*, pages 119–128, 1993.

[16] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[17] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *VLDB*, pages 537–548, 1991.

[18] J. L. Wolf, D. M. Dias, and P. S. Yu. A parallel sort merge join algorithm for managing data skew. *IEEE Trans. Parallel Distrib. Syst.*, 4(1):70–86, 1993.

[19] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. In *ICDE*, pages 200–209, 1991.

[20] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek. New algorithms for parallelizing relational database joins in the presence of data skew. *IEEE Trans. Knowl. Data Eng.*, 6(6):990–997, 1994.

[21] X. Zhou and M. E. Orlowska. Handling data skew in parallel hash join computation using two-phase scheduling. In *IEEE 1st International Conference on Algorithm and Architecture for Parallel Processing*, pages 527–536 vol.2, 1995.

APPENDIX

A. PRPD PROOF OF CORRECTNESS

This section proves the correctness of the PRPD plan. Let

$$\begin{array}{ll} R_{redis} = \bigcup_{i=1}^{n} R^i_{redis}, & S_{redis} = \bigcup_{i=1}^{n} S^i_{redis}, \\ R_{loc} = \bigcup_{i=1}^{n} R^i_{loc}, & S_{loc} = \bigcup_{i=1}^{n} S^i_{loc}, \\ R_{dup} = R^i_{dup} \forall 1 \le i \le n, & S_{dup} = S^i_{dup} \forall 1 \le i \le n. \end{array}$$

We have

$$\begin{aligned} R \bowtie S &= (R_{redis} \cup R_{loc} \cup R_{dup}) \bowtie (S_{redis} \cup S_{loc} \cup S_{dup}) \\ &= (R_{redis} \bowtie S_{redis}) \cup (R_{redis} \bowtie S_{loc}) \\ &\quad \cup (R_{redis} \bowtie S_{dup}) \cup (R_{loc} \bowtie S_{redis}) \\ &\quad \cup (R_{loc} \bowtie (S_{loc}) \cup (R_{loc} \bowtie S_{dup}) \\ &\quad \cup (R_{dup} \bowtie S_{redis}) \cup (R_{dup} \bowtie S_{loc}) \\ &\quad \cup (R_{dup} \bowtie S_{dup}) \\ &= (R_{redis} \bowtie S_{redis}) \cup (R_{loc} \bowtie S_{dup}) \cup (R_{dup} \bowtie S_{loc}) \\ &= \bigcup_{\substack{1 \le i \le n \\ 1 \le j \le n}} (R^i_{redis} \bowtie S^j_{redis}) \cup \bigcup_{1 \le i \le n} (R^i_{loc} \bowtie S_{dup}) \cup \\ &\quad \bigcup_{1 \le i \le n} (R_{dup} \bowtie S^i_{loc}) \\ &= \bigcup_{i=1}^{n} (R^i_{redis} \bowtie S^i_{redis} \cup R^i_{loc} \bowtie S^i_{dup} \cup R^i_{dup} \bowtie S^i_{loc}) \end{aligned} \tag{1}$$

This is because both R and S are split into three disjoint sets, such that

$$\begin{array}{ll} R_{redis} \cup R_{loc} \cup R_{dup} = R, & S_{redis} \cup S_{loc} \cup S_{dup} = S, \\ R_{redis} \cap R_{loc} = \varnothing, & R_{redis} \cap R_{dup} = \varnothing, \\ R_{loc} \cap R_{dup} = \varnothing, & S_{redis} \cap S_{loc} = \varnothing, \\ S_{redis} \cap S_{dup} = \varnothing, & S_{loc} \cap S_{dup} = \varnothing. \end{array}$$

Since these sets of R and S are split by the join columns, the following joins all produce the empty set:

$$\begin{array}{ll} R_{redis} \bowtie S_{loc} = \varnothing, & R_{redis} \bowtie S_{dup} = \varnothing, \\ R_{loc} \bowtie S_{redis} = \varnothing, & R_{loc} \bowtie S_{loc} = \varnothing, \\ R_{dup} \bowtie S_{redis} = \varnothing, & R_{dup} \bowtie S_{dup} = \varnothing. \end{array}$$

and $\bigcup_{\substack{1 \le i \le n \\ 1 \le j \le n}} (R^i_{redis} \bowtie S^j_{redis}) = \bigcup_{i=1}^{n} (R^i_{redis} \bowtie S^i_{redis})$ since $R^i_{redis} \bowtie S^j_{redis} = \varnothing$ for any $i \ne j$ due to the fact that any two rows hash partitioned on different PUs by their join attributes cannot match.

Equation (1) shows that we only need to do three joins on each PU though there may be three spools for R and three spools for S on each PU.