Efficient outer join data skew handling in parallel DBMS

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

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

ABSTRACT

Large enterprises have been relying on parallel database management systems (*PDBMS*) to process their ever-increasing data volume and complex queries. The scalability and performance of a PDBMS comes from load balancing on all nodes in the system. Skewed processing will significantly slow down query response time and degrade the overall system performance. Business intelligence tools used by enterprises frequently generate a large number of outer joins and require high performance from the underlying database systems. Although extensive research has been done on handling skewed processing for inner joins in PDBMS, there is no known research on data skew handling for parallel outer joins. We propose a simple and efficient outer join algorithm called OJSO (Outer Join Skew Optimization) to improve the performance and scalability of parallel outer joins. Our experimental results show that the OJSO algorithm significantly speeds up query elapsed time in the presence of data skew.

Categories and Subject Descriptors

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

General Terms

Algorithms

Keywords

data skew, outer join, parallel DBMS

1. INTRODUCTION

Parallel processing continues to be important in large data warehouses as data warehouse demand continues to expand to higher volumes, greater numbers of users, and more applications.

In a shared nothing parallel architecture [20], multiple nodes communicate via high-speed interconnect network and each node has exclusive access to its main memory and disk(s). In modern 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.

A PDBMS can easily scale up "horizontally" by adding more nodes to the system. Load balancing is critical in achieving high performance and scalability in PDBMS. Research has shown that a shared-nothing PDBMS has near linear speed-up under evenly balanced conditions [7]. On the other hand, a query processing skewed data not only drastically slows down its response time, but generates hot nodes, which become a bottleneck throttling the overall system performance. Extensive research has been done on handling skewed processing for inner joins in PDBMS [21, 8, 1, 23, 22, 24, 10, 14, 13, 6, 19, 15, 12, 26, 2, 25]. However, there is no known research on data skew handling for parallel outer joins. Modern data warehouses receive most queries from business intelligence tools, which frequently generate a large number of outer joins. For example, it is not unusual to see that a significant percentage of the queries issued by CRM tools against large data warehouses contain thirty or more outer joins in a single query. Unique to the nature of outer joins, severe skewed processing can happen in multiple outer joins on a parallel database system even when all base tables have no skewed data. From our experience with various industries, we have seen that outer joins result in serious skewed processing and can cause transactions to abort often after hours of running in large data warehouses, while inner joins on the same data sets perform efficiently without skewed processing. We propose a simple and efficient outer join algorithm called OJSO (Outer Join Skew Optimization) to improve the performance and scalability of parallel outer joins 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.

We make the following contributions in the paper:

- We propose a practical and efficient algorithm (OJSO) to handle data skew in parallel outer joins motivated by real business problems.
- The OJSO algorithm does not require major changes to the current implementation of a shared-nothing architecture and is easy to implement.

VLDB '09, August 24-28, 2009, Lyon, France

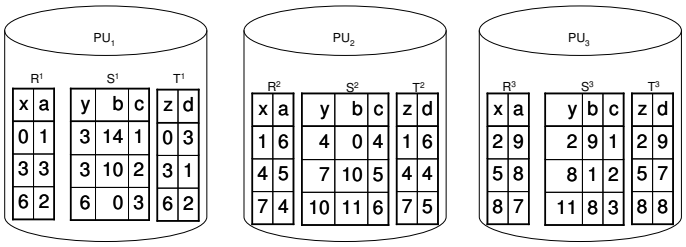

Figure 1: Three relations R, S and T are hash partitioned on a three parallel-unit system. The partitioning columns are $R.x$, $S.y$ and $T.z$ respectively. The hash function, $h(i) = i \ mod \ 3 + 1$, places a tuple with value i in the partitioning column on the $h(i)$-th PU.

- Our scalability and performance experiments show the efficiency of the proposed OJSO algorithm.

The rest of the paper is organized as follows. In Section 2 we discuss the outer join data skew problem studied in this paper. Section 3 presents the OJSO algorithm. Section 4 shows the experimental results. Section 5 discusses related work. Section 6 concludes the paper.

2. PROBLEM DESCRIPTION

2.1 Conventional outer join algorithm

Consider the following two left outer joins

$$R \underset{R.a=S.b}{\overset{o}{\bowtie_L}} S \underset{S.c=T.d}{\overset{o}{\bowtie_L}} T$$

from the following query.

```
select x, y, z, a, c
from R left outer join S on R.a=S.b
       left outer join T on S.c=T.d    (Query 1)
```

Relations on a shared-nothing PDBMS 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 (often the first few columns) if none is designated by the user.

As an example, Figure 1 shows the partitioning of the three relations $R(x, a)$, $S(y, b, c)$ and $T(z, d)$ in Query 1 on a three-PU system, assuming that the partitioning columns are $R.x$, $S.y$ and $T.z$ for R, S and T 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 $(x = 0, a = 1)$ of R is placed on the first PU since $h(0) = 1$. The fragment of R, S and T on the i-th PU is denoted as R^i , S^i and T^i respectively.

Assume that the optimizer chooses to left outer join R and S, then left outer join T. The two outer joins in Query 1 are typically evaluated by the following three-step conventional algorithm.

Assume there are n PUs in the system. The following three steps are performed in parallel on all PUs.

- Step 1
  - 1) Rows of R are redistributed based on the hash values of its join attribute $R.a$ and are stored in a temporary table R_{redis}.
  - 2) Rows of S are redistributed based on the hash values of its join attribute $S.b$ and are stored in a temporary table S_{redis}.

  The redistribution in Step 1.1 and Step 1.2 is often simply called *hash redistribution*. Step 1 sends matching rows of R and S to the same PUs in preparation for the actual joins. Notice that rows in R^i_{redis} $(1 \le i \le n)$ have potential matching rows only from S^i_{redis} on the same PU and have no matching rows from any other S^j_{redis} where $i \ne j$. Figure 2 shows the result of Step 1 on the example data in Figure 1.

- Step 2
  - 1) Rows of R_{redis} and S_{redis} are left outer joined and the results are stored in a temporary table J. Figure 3 shows the results of the left outer join. Rows from the results of the left outer join (J) are hash redistributed on the join attribute J.c[1] and the results are stored in a temporary table J_{redis}.
  - 2) Rows of T are redistributed based on the hash values of its join attribute $T.d$ and are stored in a temporary table T_{redis}.

  Figure 4 shows the result of hash redistributing J and T on $J.c$ and $T.d$ respectively, assuming that the system hash redistributes *null*s to the first PU (i.e., $h(null) = 1$).

- Step 3

  Rows of J_{redis} and T_{redis} are left outer joined and the results are stored in a temporary table F. The final results of the two outer joins are shown in Figure 5.

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 sub-steps 2.1 and 2.2 in the second step.

2.2 Data skew in Outer joins

Notice that the three tables R, S and T in Figure 1 are all evenly partitioned across three PUs. Figure 2 shows that the redistribution in Step 1 is also evenly balanced. Figure 3 shows that the first left outer join produces the same number of rows on each PU, causing even processing on all three PUs. However, skewed processing happens in Step 2.1, as Figure 4 shows that the number of rows the first PU receives from the results of the first left outer join is 7 times that of any other PU, which creates skewed processing in the system. The skewed processing happens because all *dangling*

[1]In practice, each row from the join result is immediately hash redistributed after it is computed (though the underlying messaging system may choose to buffer a few rows and send them in bulk).

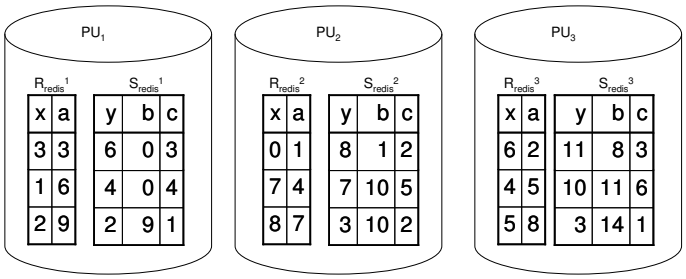

Figure 2: The result of hash redistributing R and S on their join attributes ($R.a$ and $S.b$) to two temporary tables R_{redis} and S_{redis}.

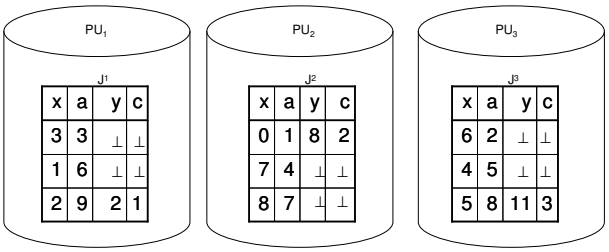

Figure 3: The results of left outer joining R_{redis} and S_{redis} (R_{redis} and S_{redis} are shown in Figure 2) are stored in a temporary table J.

rows of R [2] padded with nulls from the first outer join are hash redistributed to the first PU (which is called a *hot PU*).

In our experience with various industrial applications, it is not uncommon that more than half of the rows in a large table R have no matching rows in S, causing severe skewed processing in later joins.

Adding more nodes to the system will not solve the skew problem because all dangling rows from the results of the first outer join will still be sent to a single PU. In fact, adding more nodes (for the same data sets) will make each non-hot PU colder (having fewer rows) and make the hot PU comparatively even hotter.

In contrast, if we change the outer joins to inner joins in Query 1 as shown in Query 2, then there is no skewed processing in evaluating $R \overset{R.a=S.b}{\bowtie} S \overset{S.c=T.d}{\bowtie} T$.

```
select x, y, z, a, c
from R inner join S on R.a=S.b
       inner join T on S.c=T.d    (Query 2)
```

The three-step algorithm described in Section 2.1 is slightly modified to evaluate $R \overset{R.a=S.b}{\bowtie} S \overset{S.c=T.d}{\bowtie} T$. The only changes are that we change the join method from outer join to inner join in Step 2.1 and Step 3. Figure 2 shows the result of Step 1. Figure 6 shows the result of Step 2.1 (inner joining R and S). Figure 7 shows the result of Step 2 (hash redistributing the results of the first inner join and T). For completeness, Figure 8 shows the final result of the two inner joins (Step 3). Figures 2, 6, 7 and 8 show that every step has even processing on all three PUs for Query 2.

[2]Dangling rows of R are rows of R having no matching rows in S.

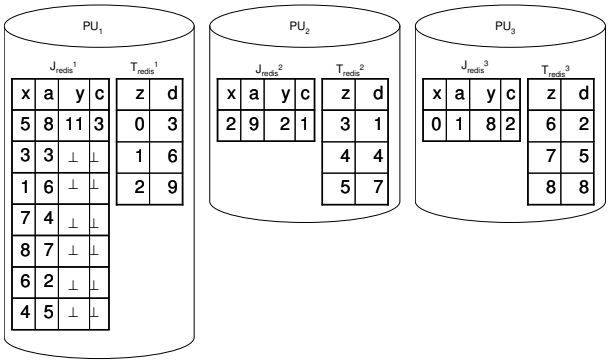

Figure 4: The result of hash redistributing J (shown in Figure 3) and T (shown in Figure 1) on their join attributes ($J.c$ and $T.d$) to two temporary tables J_{redis} and T_{redis}.

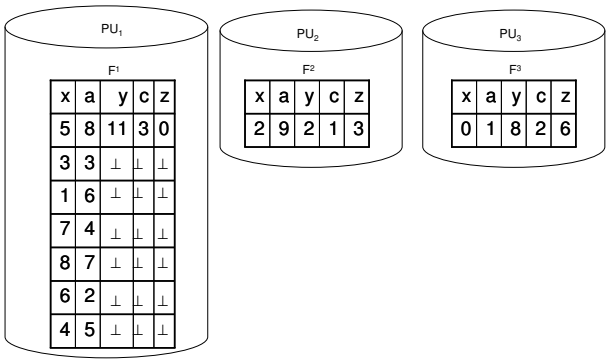

Figure 5: The final results of the two outer joins in Query 1 are stored in a temporary table F.

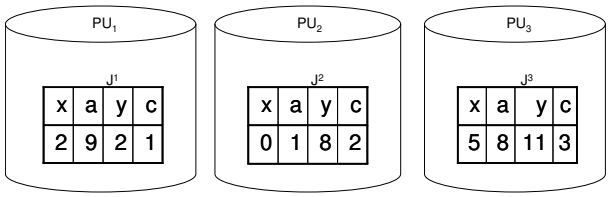

Figure 6: The results of the first inner join ($R_{redis} \overset{R_{redis}.a=S_{redis}.b}{\bowtie} S_{redis}$, R_{redis} and S_{redis} are shown in Figure 2) are stored in a temporary table J.

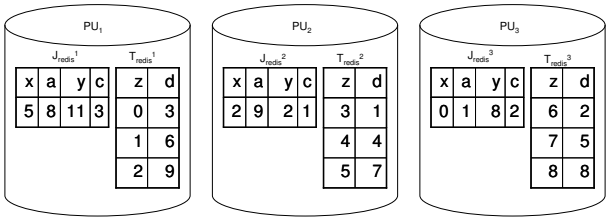

Figure 7: The result of hash redistributing J (shown in Figure 6) and T (shown in Figure 1) on their join attributes ($J.c$ and $T.d$) to two temporary tables J_{redis} and T_{redis}.

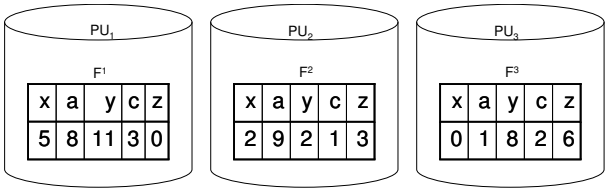

Figure 8: The final results of the two inner joins in Query 2 are stored in a temporary table F. F^i denotes the fragment of F on the i-th PU.

3. ALGORITHM

We now present our algorithm called OJSO (Outer Join Skew Optimization) to handle the data skew problem in outer joins described in Section 2. The basic idea is to treat dangling rows of R and non-dangling rows of R differently in the second step in evaluating the first outer join: we keep dangling rows of R locally on every PU instead of hash redistributing them to a single PU, and hash redistribute the rest of the results of the first outer join.

Assume there are n PUs in the system. The following steps are executed in parallel on every PU for Query 1 according to the OJSO algorithm.

- Step 1
  - 1) Rows of R are redistributed based on the hash values of its join attribute $R.a$ and are stored in a temporary table R_{redis}.
  - 2) Rows of S are redistributed based on the hash values of its join attribute $S.b$ and are stored in a temporary table S_{redis}.

  Figure 2 shows the result of Step 1 on the example data in Figure 1. This step is the same as the first step in the conventional algorithm presented in Section 2.

- Step 2
  - 1) Rows of R_{redis} and S_{redis} are left outer joined and the results are split into two temporary tables J_{2redis} and J_{local}. J_{2redis} contains rows created from matching rows from R_{redis} and S_{redis} while J_{local} contains rows created from dangling rows of R_{redis} padded with nulls. Figure 9 shows the result of splitting the results of the first left outer join on every PU.

    J_{2redis} are hash redistributed on the column $J_{2redis}.c$ and the results are stored in a temporary table J_{redis}. Notice that in practice the temporary table J_{2redis} is only logical (not materialized) since every row in J_{2redis} is hash redistributed on the fly after it is computed.

    Rows in J_{local} are kept locally and padded with nulls for the projected attribute(s) of T. The results are stored in a temporary table $J_{locpadding}$.
  - 2) Rows of T are redistributed based on the hash values of its join attribute $T.d$ and are stored in a temporary table T_{redis}.

  Figure 10 shows the result of Step 2.

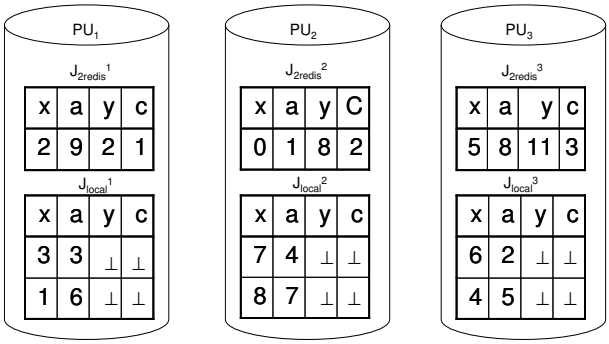

Figure 9: The results of left outer joining R_{redis} and S_{redis} are split into two temporary tables J_{2redis} and J_{local}.

- Step 3

  Rows of J_{redis} and T_{redis} are left outer joined and the results are stored in a temporary table F_{redis}.

- Step 4

  The final results of the two outer joins are the union of F_{redis} and $J_{locpadding}$ [3], as shown in Figure 11.

A visual description of the steps in the OJSO algorithm is shown in Figure 12.

As shown from Figures 1, 2, 9, 10 and 11, no skewed processing occurs in the query processing. Notice that the OJSO algorithm works in the same way if the join attributes $S.c$ and $T.d$ are multiple columns or are the same attribute(s). The correctness of the algorithm comes from the fact that dangling rows in the results of the first outer join will have no matching rows in the second outer join. Thus dangling rows in the results of the first outer join can be correctly and efficiently kept locally and padded with nulls as part of the final results.

Although the example query (Query 1) shows only two sequential left outer joins, the OJSO algorithm applies to both right outer joins and full outer joins and is repeatedly applied to process more than two outer joins. In implementation the OJSO algorithm handles a single outer join at a time. After join order planning is done, the optimizer goes through all outer joins in a query, analyzes what join columns whose values could be set to NULLs and are used in later outer joins, and then repeatedly calls the OJSO algorithm to execute each outer join with the instruction of whether the results of an outer join should be split based on its previous analysis on join columns. Since the OJSO algorithm handles a single outer join at a time and is repeatedly called by the optimizer, the OJSO algorithm applies to arbitrary join graphs not just to a "chain" of outer joins as shown in the example in Query 1.

4. EXPERIMENTAL EVALUATION

In this section, we compare the scalability and performance of the OJSO algorithm described in Section 3 and the conventional algorithm described in Section 2.1.

The test system we use for the experiments has 8 nodes. Each node has 4 Pentium IV 3.6 GHz CPUs, 4 GB memory,

[3]The two relations have no rows in common. Thus the union is only logical, requires no expensive duplicate removal.

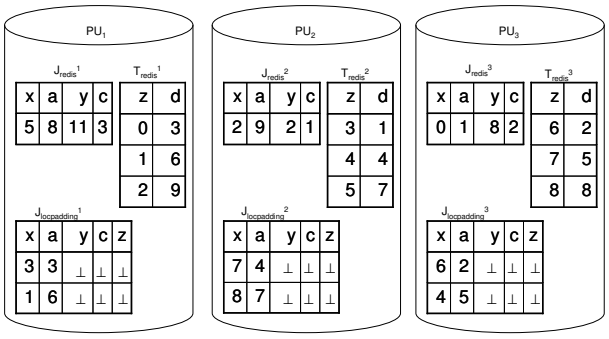

Figure 10: The result of hash redistributing J_{2redis} (shown in Figure 9) and T (shown in Figure 1) on their join attributes to two temporary tables J_{redis} and T_{redis}. $J_{locpadding}$ is created from J_{local} (shown in Figure 9) with padded nulls.

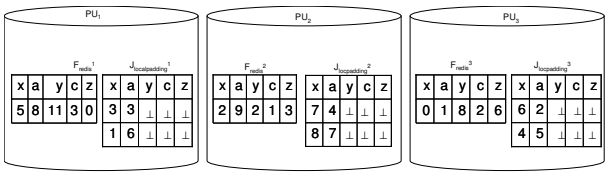

Figure 11: The results of the second outer join in Query 1 are stored in a temporary table F. The final result for Query 1 is the union of F and $J_{locpadding}$.

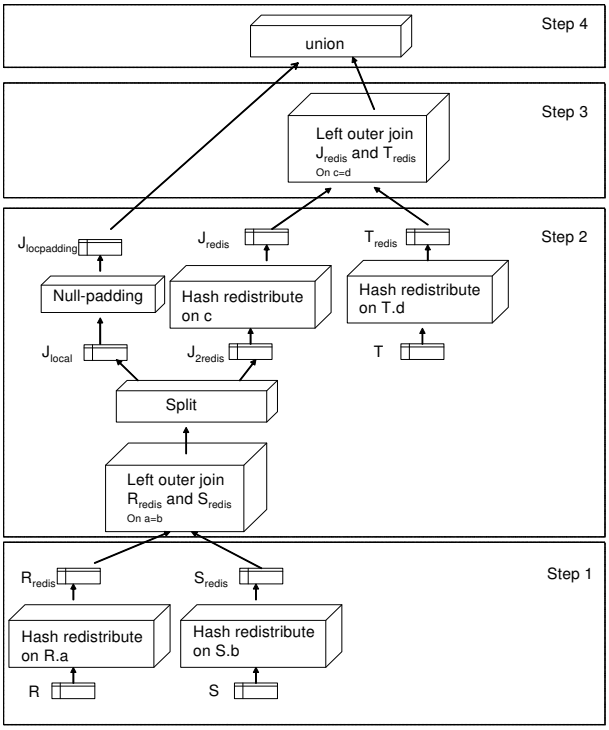

Figure 12: Visual presentation of the OJSO algorithm.

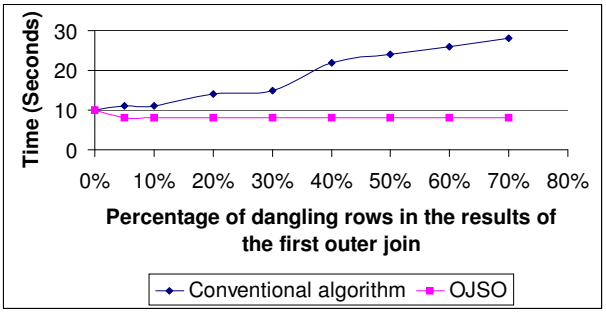

Figure 13: Query execution time on 8 nodes (16 Parallel Units) system. Each table (R, S and T) has 10 million rows.

and 2 dedicated 146 GB hard drives. Each node is configured to run 2 PUs to take advantage of the two hard drives. In the first experiment, we generate 10 million rows for each table (R, S and T) and run Query 1 (in Section 2.1). We vary the percentage of dangling rows in the results of the first outer join from 0% to 70% by controlling the values in $R.a$ while keeping the sizes of R, S and T at 10 million rows each. Figure 13 shows the execution times of the OJSO algorithm and the conventional algorithm. When the percentage of dangling rows in the results of the first outer join increases, the execution time of the conventional algorithm grows almost linearly because all null values are hash redistributed to one PU and that PU becomes the system bottleneck while the execution time of the OJSO algorithm essentially stays the same.

In the second experiment, we generate 50 million rows for each table and execute Query 1 as in the first experiment. The query execution times of the OJSO algorithm and the conventional algorithm are shown in Figure 14. Again, the execution time of the conventional algorithm grows almost linearly as the percentage of dangling rows in the results of the first outer join increases while the execution time of the OJSO algorithm decreases slightly. The OJSO algorithm is faster on more "skewed data" because it keeps more data locally in Step 2 (Section 3) and reduces the redistribution cost in preparation for the second outer join.

In the rest of our experiments, we repeat the second experiment on two smaller configurations of the same test system: 2 nodes (total 4 PUs) configuration and 4 nodes (total 8 PUs) configuration. The results are shown in Figures 15 and 16 respectively. We see the same performance relationship between the percentage of dangling rows in the results of the first outer join and the execution times of the two algorithms.

5. RELATED WORK

Extensive research has been done on handling data skew in parallel inner joins. [21] categorizes four types of skew: *tuple placement skew*, *selectivity skew*, *redistribution skew*, and *join product skew*. Tuple placement skew happens when initial distribution of tuples of a table varies significantly between partitions, which can be avoided with a good hash function and proper choices of partitioning columns. In industrial deployment, partitioning columns are carefully chosen by DBAs so that tuples of large relations are most likely evenly partitioned on all PUs (parallel units). Thus, tuple

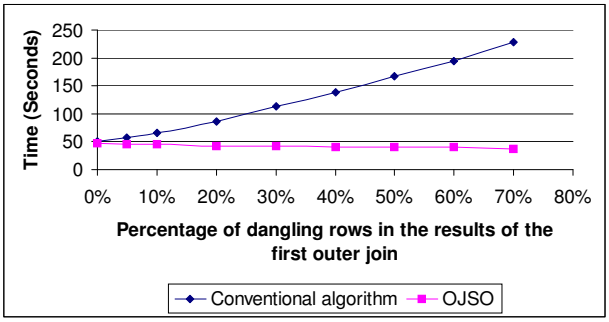

Figure 14: Query execution time on 8 nodes system (16 Parallel Units). Each table (R, S and T) has 50 million rows.

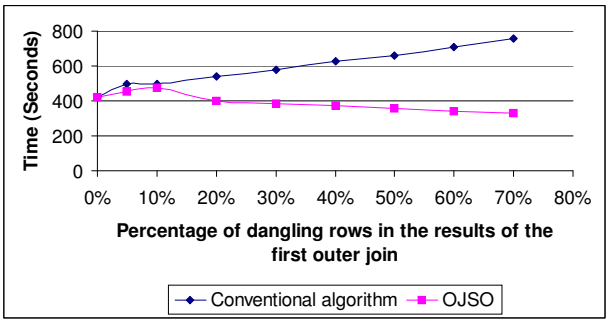

Figure 15: Query execution time on 2 nodes system (4 Parallel Units). Each table (R, S and T) has 50 million rows.

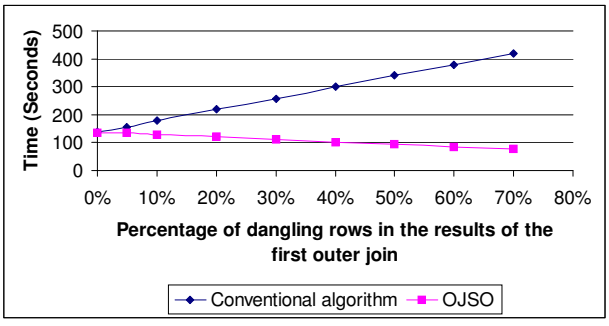

Figure 16: Query execution time on 4 nodes system (8 Parallel Units). Each table (R, S and T) has 50 million rows.

placement skew is not an issue in practice and nearly no work focuses on it. Selectivity skew is caused by different selectivity of selection predicates. Selectivity skew becomes a problem only when it causes redistribution skew or join product skew. Most prior work does not consider selectivity skew either. Redistribution skew occurs when PUs receive different number of tuples when they are redistributed in preparation for joins. Join product skew occurs when the join selectivity at each node differs, causing imbalance in the number of tuples produced at each PU. Redistribution skew and join product skew are closely related problems and have been extensively studied.

Prior algorithms handling redistribution skew and join product skew can be roughly classified into the following two categories. The algorithms in the first category are static in the sense that they need to detect the presence of data skew before they are applied [8, 15, 13, 23, 22, 24, 1, 6, 25]. The algorithms in the second category handle data skew dynamically. The basic idea is that 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 [19, 12, 26, 15, 13, 2].

The outer join data skew problem studied in this paper can be categorized as redistribution skew according to [21]. However, prior algorithms in the first category cannot be adapted to handle the outer join skew problem presented in Section 2 due to the nature of pipelined execution of the steps described in Section 2 and the static nature of the first category of algorithms. Notice that the first outer join in Query 1 (Section 2) does not create join product skew since every PU produces the same number of rows (Figure 3), making it difficult to detect data skew at each PU locally. In principle, prior algorithms in the second category can be adapted to handle the outer join skew problem because they can handle skewed processing dynamically. However, to our best knowledge, no effective dynamic skew handling mechanism has been implemented by any major parallel DBMS vendors, either because of their high implementation complexity or communication cost, or the significant changes required to the current shared-nothing architecture. The OJSO algorithm proposed in this paper is efficient in handling the outer join data skew problem by being tuned for outer joins.

Research has also been done on other aspects of optimizing outer joins including outer join elimination [9, 5, 4, 3] and view matching for outer join views [16, 17, 18, 11]. However, none of the above work handles data skew in outer joins.

6. CONCLUSIONS

One of the important challenges in PDBMS is to evenly balance workload among all nodes in the system. Based on our observations of PDBMS deployment at various industries, we notice that outer joins pose unique challenges and opportunities in skewed processing. Motivated by the outer join skew problem that arises in many business applications, we propose a simple and efficient algorithm called OJSO (Outer Join Skew Optimization) to prevent skewed processing in outer joins. One big advantage of the OJSO algorithm is that it is easy to implement and does not require major changes to the implementation of the shared-nothing architecture, since it does not require any new type of central coordination or communication among parallel units. Our ex-

periments in scalability and performance have demonstrated the effectiveness of the OJSO algorithm in improving query execution time.

7. REFERENCES

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

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

[3] G. Bhargava, P. Goel, and B. R. Iyer. Simplification of outer joins. In *CASCON '95: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 1995.

[4] G. Bhargava, P. Goel, and B. R. Iyer. Efficient processing of outer joins and aggregate functions. In *ICDE*, pages 441–449, 1996.

[5] A. L. P. Chen. Outerjoin optimization in multidatabase systems. In *DPDS '90: Proceedings of the second international symposium on Databases in parallel and distributed systems*, pages 211–218, New York, NY, USA, 1990. ACM.

[6] 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.

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

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

[9] A. Ghazal, A. Crolotte, and R. Bhashyam. Outer join elimination in the teradata rdbms. In *DEXA*, pages 730–740, 2004.

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

[11] A. Gupta, H. V. Jagadish, and I. S. Mumick. Maintenance and self maintenance of outer-join views. In *NGITS*, pages 0–, 1997.

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

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

[14] 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.

[15] 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.

[16] P.-Å. Larson and J. Zhou. View matching for outer-join views. In *VLDB*, pages 445–456, 2005.

[17] P.-Å. Larson and J. Zhou. Efficient maintenance of materialized outer-join views. In *ICDE*, pages 56–65, 2007.

[18] P.-Å. Larson and J. Zhou. View matching for outer-join views. *VLDB J.*, 16(1):29–53, 2007.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD Conference*, pages 1043–1052, 2008.

[26] 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.