IMPROVING JOIN PERFORMANCE FOR SKEWED DATABASES

Bryce Cutt, Ramon Lawrence

University of British Columbia Okanagan
brycec@interchange.ubc.ca, ramon.lawrence@ubc.ca

ABSTRACT

The largest queries in data warehouses and decision support systems use hybrid hash join to relate information in multiple tables. Hybrid hash join functions independently of the data distributions of the join relations. Real-world data sets are not uniformly distributed and often contain significant skew. Although partition skew has been studied for hash joins, no prior work has examined how exploiting data skew can improve performance. In this paper, we present histojoin, a join algorithm that uses histograms to identify data skew and improve join performance. Experimental results show that for skewed data sets histojoin performs significantly fewer I/O operations and is faster by 20 to 60% than hybrid hash join.

Index Terms— hash join, skew, histogram, data warehouse

1. INTRODUCTION

Hybrid hash join is the standard join used in database systems to process large join queries. These queries occur in data warehouses and decision support systems and may take minutes to execute. Any performance improvement is significant due to the cost and prevalence of hash-based joins. Hybrid hash join does not adapt its operation in response to skew in the join relations. Real data sets often contain skew. Many data sets follow the "80/20 rule" where a small subset of the data items occur much more frequently (e.g. top selling items, best customers, etc.).

In this paper, we present a modification to hybrid hash join that exploits data skew to improve performance for one-to-many (primary-to-foreign key) joins. The basic idea is to buffer in memory the tuples of the primary key relation whose key values occur most frequently in the foreign key relation. Keeping tuples in memory that have a higher probability of joining with other tuples reduces the number of I/Os performed. The high frequency tuples are detected by using histograms. Histograms are commonly produced by database systems for query optimization and can be exploited at no cost by the join algorithm.

978-1-4244-1643-1/08/$25

The contributions of this paper are as follows:

- An analysis of the advantage of exploiting data skew to improve hybrid hash join performance.
- A modification of hybrid hash join called *histojoin* that uses a histogram to detect data skew and adapts its memory allocation to maximize its performance.
- An experimental evaluation that demonstrates the benefits of histojoin for TPC-H queries.

The organization of this paper is as follows. In Section 2, we describe previous and related work. An overview of the approach is in Section 3. Implementation details of histojoin are in Section 4. Experimental results are in Section 5, and the paper closes with future work and conclusions.

2. PREVIOUS WORK

Consider two relations $R(\underline{A})$ and $S(\underline{B}, A)$ where attribute A is the join attribute between R and S. Assume that the number of tuples of R, denoted as $|R|$, is smaller than the number of tuples of S (i.e. $|R| < |S|$).

Hybrid hash join [1] works by partitioning the inputs with a hash function on the join attribute(s). Tuples in each relation will only join if they fall in the same partition after hashing. Hybrid hash join uses any additional memory beyond what is needed for partitioning to store one partition in memory of the smaller relation (R). While the larger relation (S) is being partitioned, any tuples that fall into the in-memory partition can be immediately joined and output. Thus, there is an advantage to keeping as much of the smaller relation in memory as possible as this avoids writing tuples of the smaller relation and the matching tuples of the larger relation to disk.

Dynamic hash join (DHJ) [2, 3] is similar to hybrid hash join except that it allows the partition sizes to vary during execution. Instead of picking only one partition to remain memory resident before the join begins, DHJ allows all partitions to be memory resident initially and then flushes partitions as required when memory is full. Although DHJ adapts to changing memory conditions, there has been no research on determining what is the best partition to flush to maximize

performance. Various approaches select the largest or smallest partition, a random partition, or use a deterministic ordering. No approach has considered using data distributions to determine the optimal partition to flush.

If the data is skewed such that certain tuples in R join to many more tuples in S than the average, it is preferable that those tuples of R remain in memory. Skew can be classified [4] as either *partition skew* or *intrinsic data skew*. Partition skew is when the partitioning algorithm constructs partitions of non-equal size (often due to intrinsic data skew but also due to the hash function itself). Minimizing partition skew has been considered for distributed databases [5] and DHJ [3, 6]. Handling partition skew is orthogonal to this work. Intrinsic data skew is when data values are not distributed uniformly. For primary-to-foreign key joins, data skew causes primary key values to occur with different frequencies in the foreign key relation. No previous join has exploited intrinsic data skew by keeping the frequently occurring values in memory.

Our histojoin algorithm works with any histogram method implemented in the DBMS. An overview of histograms can be found in [7]. The actual construction of the histograms is orthogonal to this work as our operator uses pre-existing histograms and does not construct or maintain them directly.

3. GENERAL APPROACH

The general approach is to use the extra memory available to the hash join to buffer the tuples that will participate in the most join results. Consider a primary-to-foreign key join between $R(\underline{A})$ and $S(\underline{B}, A)$ on A, where R is the smaller relation and some subset of its tuples are buffered in memory. Unlike hybrid hash join that selects a random subset of the tuples of R to buffer in memory, the tuples buffered in memory will be chosen based on the values of A that are the most frequently occurring in relation S.

For example, let R represent a $Product$ table, and S represent a $Lineitem$ table. Every company has certain products that are more commonly sold than others. A common product may be associated with thousands of line items and a rare product only a handful. If a single product tuple ordered thousands of times is kept in memory when performing the join, every matching tuple in $Lineitem$ does not need to be written to disk and re-read during the cleanup phase.

Hash partitioning randomizes tuples in partitions. This is desirable to minimize the effect of partition skew, but data skew is also randomized. Hybrid hash join has no ability to detect data skew in the probe relation or exploit it by intelligent selection of in-memory partitions.

Our approach uses two levels of partitioning. The first level performs range partitioning where ranges of join values of R are selected to be memory-resident. Tuples that do not fall into the ranges are partitioned using a hash function as usual. The memory partition size is bounded by the memory size available to the join. The operation of the algorithm is identical to hybrid hash join except that only the pre-determined partition is memory-resident, and there must be a range check performed for each tuple to determine if it belongs in the in-memory partition or not.

3.1. Theoretical Performance Analysis

There is a significant performance advantage of using skew-aware partitioning versus random partitioning (hybrid hash join) for skewed relations. Let f represent the fraction of the smaller relation (R) that is memory resident: $f = M/|R|$, where M is the memory size (maximum partition size). The number of I/O operations performed by hybrid hash join is $3 * (1 - f) * (|R| + |S|)$. Let g represent the fraction of the *larger* relation (S) that joins with the in memory fraction f of R. If the distribution of the join values in S is uniform, then $f = g$. Data skew allows $g > f$ if memory resident tuples are chosen properly. The number of I/Os performed is $3 * (1 - f) * |R| + 3 * (1 - g) * |S|$. The percentage difference in I/Os between skew-aware and random partitioning is $\frac{3*(1-f)*(|R|+|S|)-3*((1-f)*|R|+(1-g)*|S|)}{3*(1-f)*(|R|+|S|)} = \frac{(g-f)*|S|}{(1-f)*(|R|+|S|)}$.

The percentage difference in total I/Os performed is directly proportional to the ratio of $|R|$ and $|S|$ and the difference between f and g. The difference between f and g is bounded by the intrinsic skew in the data set and is limited by how we exploit that skew during partitioning.

As an example, let $f = 0.2$ and $g = 0.8$ with $|R| = |S|$. That is, there is sufficient memory to store 20% of the smaller relation in memory, but due to data skew that 20% matches with 80% of the larger relation ("80/20 rule"). Then, the skew-aware join algorithm will perform 38% fewer I/Os than standard hash join. If S is four times larger than R, then skew-aware join will be 60% faster than hybrid hash join.

4. HISTOJOIN ALGORITHM

A low cost technique for performing skew-aware partitioning is by using histograms. Histograms [7] are used in all commercial databases for query optimization and provide an approximation of the data distribution of an attribute. A histogram divides the domain into ranges and calculates the frequency of the values in each range. An example histogram produced by Microsoft SQL Server 2005 for the TPC-H relation *Lineitem* on attribute *partkey* is in Figure 1.

The advantage of using histograms is that they are readily available from the DBMS, calculated and maintained external to the join algorithm, and require no modification to the query optimizer or join algorithm to use. On examination of the histogram, the query optimizer can determine if the histojoin algorithm will be beneficial.

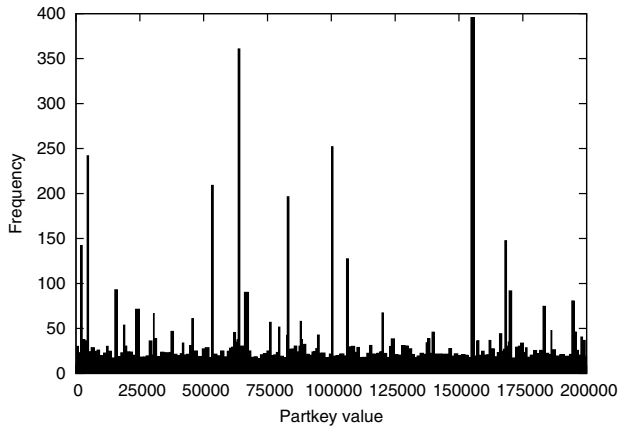

Fig. 1. Partkey Histogram for Lineitem Relation TPC-H 1 GB Zipf Distribution (z=1)

4.1. Selecting In-Memory Tuples

Given a histogram that demonstrates skew, histojoin must determine a set of join attribute range(s) that constitute the most frequently occurring values in S. Tuples of R with these frequently occurring values are the ones buffered in memory. For instance in Figure 1 there are several ranges of partkey that occur frequently in *LineItem*. Our greedy algorithm reads the histogram for S on the join attribute, sorts its buckets by frequency, and selects as many buckets that fit into memory in the order of highest frequency first. The steps are:

- Assume each histogram bucket entry is a 4-tuple of the form *(MAXVAL, COUNT, EQROWS, DISTROWS)*.[1] *MAXVAL* is the upper value defining the histogram range. *COUNT* is the number of rows in the relation with a value in the range of this histogram bucket. *EQROWS* is the number of rows in the relation whose value is exactly equal to *MAXVAL*. *DISTROWS* is the distinct number of values in R in the histogram bucket range. [2]
- A bucket *frequency* is calculated as: $(COUNT + EQROWS)/DISTROWS$.
- Sort the buckets in decreasing order by frequency.
- The sorted list is traversed in order. Assume the size of memory in tuples is M, and $count$ is the number of tuples currently in the in-memory partition. A histogram bucket range is added to the in-memory partition if $count + DISTROWS <= M$.
- The previous step is repeated until the histogram is exhausted, there is no memory left to allocate, or the current bucket does not fit entirely in memory.

Consider the histogram in Figure 2, and a join memory size of 400 tuples. The first histogram bucket added has range 751-1000 (250 tuples) as its weight is 4.6. The second histogram bucket added has range 101-200 with weight 3.1. The remaining memory available can be allocated in various ways: leave as overflow, find next bucket that fits, or divide a bucket. With integer values, it is possible to take the next best bucket and split the range. In this case, the range from 1-100 can be divided into a subrange of 1-50.

MAXVAL	COUNT	EQROWS	DISTROWS	FREQ
100	300	5	100	3.05
200	300	10	100	3.1
350	150	100	150	1.67
500	200	40	150	1.6
750	249	1	250	1
1000	650	500	250	4.6

Fig. 2. Histogram Partitioning Example

One optimization with maxdiff histograms is that the histogram breaks out the frequency of occurrence of the upper range value ($MAXVAL$). Note in the example that value 350 occurs 100 times even though on average the other values in the range of 201-349 only occur once. The tuple with value 350 should be memory resident. Our algorithm creates separate one value ranges for each separation value. When sorted, these ranges may be selected independently of the rest of their histogram bucket. For example, with a memory size of 400 tuples, our algorithm selects the following ranges: 1000, 350, 500, 200, 100, 751-999, 101-199, 1-47. (The last range is a partial range of 1-100.) A tuple is in the in-memory partition if it falls in one of these ranges.

4.2. Range Check

A *range check* is performed for each tuple by comparing its join attribute value with the ranges calculated in the previous step. As an optimization, adjacent ranges for integer values are merged before performing the check. In our example, the ranges would be 1-47, 100-200, 350, 500, 751-1000.

For efficiency, the range check is implemented in two ways. The first way sorts the ranges and performs a binary search using a given value to find if it is any of the ranges. The second approach for integer values uses a bit array. Consider a bit array of size 1000. A bit is set in the array if that index value should be in memory. For instance, bit 500 should be set as 500 is an in-memory value. For ranges such as 100-200, the bit range from 100 to 200 is set in the bit map. A range check using a bit array amounts to hashing the input value and checking if the bit is set in the bit array.

[1] This description is consistent with the form of maxdiff histograms in SQL Server, but simplified for presentation.

[2] The histogram provides the number of distinct values of S in the bucket which may underestimate the number of values of R in the range. A correction factor is used except for integer keys where the distinct values are calculated exactly using the bucket low and high range values.

4.3. Partitioning

Partitioning occurs similar to hybrid hash join except a range check is performed for each tuple to determine if it is in the in-memory partition. When partitioning the build relation R if the tuple is in the range, it is placed in the in-memory partition, otherwise the join attribute is hashed as usual, placed in the correct hash partition, and flushed to disk. Tuples of R in memory are organized in a chained hash table consisting of multiple buckets, each of which stores a linked list of tuples.

When partitioning the probe relation S, the range check is performed. If the tuple is in the range, it probes the in-memory partition by hashing its join attribute to find a bucket in the chained hash table. The probe tuple of S is discarded after the probe. If the tuple of S is not in the range, it is hashed to a partition, then flushed to disk. After all of R and S is partitioned, a partition of R is loaded into memory and probed with the corresponding on disk partition of S. This is repeated for all on disk partitions of R and S.

5. EXPERIMENTAL RESULTS

The histojoin algorithm was tested with the TPC-H data set. We used the TPC-H generator produced by Microsoft Research [8] to generate skewed TPC-H relations. Skewed TPC-H relations have their attribute values generated using a Zipf distribution, where the Zipf value (z) controls the degree of skew. The data sets we tested were of scale 1 GB and labeled as skewed (z=1) and high skew (z=2).

The dynamic version [2] of hybrid hash join [1] (DHJ) was compared to histojoin. Both algorithms were implemented in Java and used the same data structures and hash algorithms. The only difference between the implementations is that histojoin allocated its in-memory partition using a histogram and dynamic hash join flushed partitions to free memory without regard to data distributions.

The data files were loaded into Microsoft SQL Server 2005 and histograms generated. The histograms were exported to disk, and the data files converted to binary form. Data files were loaded from one hard drive and a second hard drive was used for temporary files produced during partitioning. The experimental machine was an Intel Pentium IV 3.0 GHz with 4 GB memory running Windows XP and Java 1.6. All results are the average of several runs. The joins tested were *LineItem-Part* on *partkey* and *LineItem-Supplier* on *suppkey*. Memory fractions, f, tested ranged from 10% to 100%.

The *LineItem-Part* results by total I/Os (includes cost of reading each relation) and by time for the z=1 data set are in Figures 3 and 4 respectively. Histojoin performs approximately 25% fewer I/O operations which results in it being about 25% faster overall. This is a major improvement for a standard operation like hash join. This improvement occurs over all memory sizes until full memory is available.

For the z=2 data set, the performance difference is even more dramatic. Histojoin performs 60% fewer I/Os resulting in 60% faster execution. The results by time are in Figure 5.

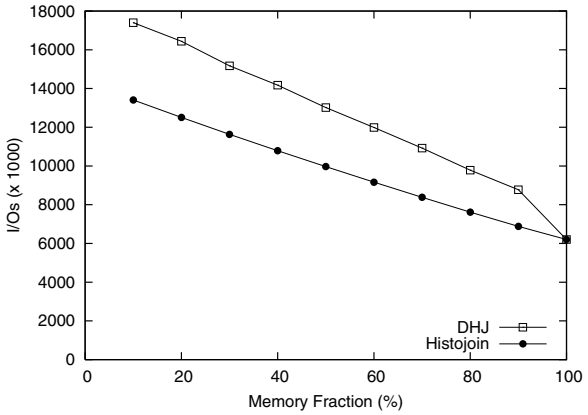

Fig. 3. Total I/Os for Lineitem-Part Join (z=1)

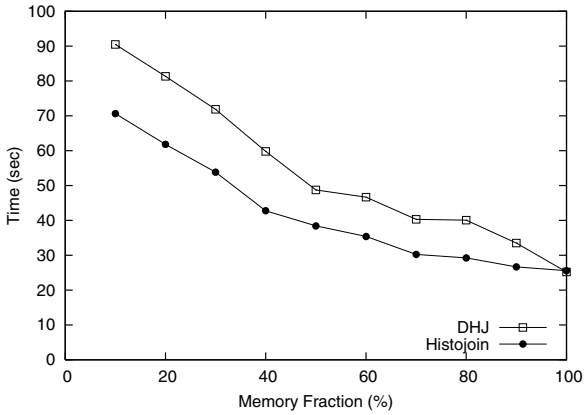

Fig. 4. Total Time for Lineitem-Part Join (z=1)

Hybrid hash join is slower because random partitioning causes the most important tuples to be distributed across all partitions. Regardless what partitions are flushed (or conversely what partition(s) remain in memory), hash join is guaranteed to not keep in memory all of the most beneficial tuples. Even worse, for highly skewed data sets, it is very likely that it will evict the absolute best partition. For instance, with 10% memory and 10 partitions, hash join has only a 10% probability of keeping the partition in memory with the value that is most frequently occurring. For the z=2 data, it happens that hash join selects the best partition as the first partition to evict from memory. Consequently, it sees little benefit even up to 90% memory (9 of the 10 partitions in memory).[3] The performance of dynamic hash join is *unpredictable* for skewed relations and is highly dependent on the partition flushing policy.

Histojoin also shows a major improvement over hash join when joining *LineItem-Supplier*. For the z=1 data set, his-

[3]The DHJ implementation is deterministic. Random flushing would improve average performance over multiple runs.

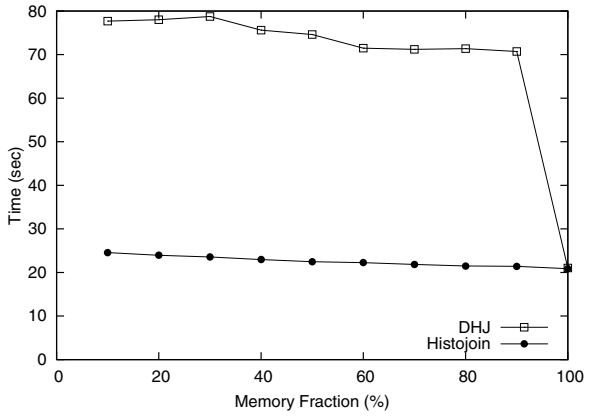

Fig. 5. Total Time for Lineitem-Part Join (z=2)

tojoin performs about 15-20% fewer total I/Os and executes 15-20% faster. For the z=2 data set, histojoin performs between 40-60% fewer total I/Os and executes 40-60% faster. A summary of the percentage total I/O savings of histojoin versus hybrid hash join for all joins is in Figure 6.

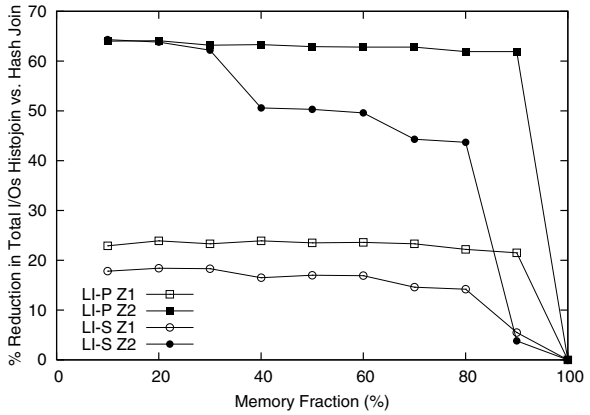

Fig. 6. % Improvement in I/Os for Histojoin vs. Hash Join

Experiments with uniform data show that the performance of histojoin and hash join is identical, as there are no tuples that occur more frequently than any other, and the performance is independent of the tuples buffered in memory.

5.1. Results Summary

For skewed data sets, histojoin dramatically outperforms hybrid hash join by 20 to 60%. This is significant because hybrid hash join is a very common operator used for processing the largest queries. As the amount of skew increases, the relative performance improvement of histojoin increases.

Histojoin introduces no performance penalty compared to hybrid hash join for uniform data sets or data sets where the skew is undetected due to selection conditions or stale histograms. Histojoin's performance improvement depends on the amount of skew detected (as given by the formula in Section 3.1). The more accurate the estimation of the distribution of the probe relation, the better the performance. Thus, histojoin will exploit whatever skew is detectable and fall back to hybrid hash join behavior otherwise.

6. CONCLUSIONS

Intrinsic data skew is prominent in databases, and hybrid hash join does not handle it well. By using pre-existing histograms on join attributes of the probe relation, it is possible to *improve* performance by using knowledge of the data distribution to identify which tuples of the build relation should be memory-resident. Histojoin has significantly better performance than DHJ (from 20 to 60%) for skewed data. Future work includes extending histojoin for many-to-many joins and multi-way joins.

7. REFERENCES

[1] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," in *SIGMOD*, 1984, pp. 1–8.

[2] D. DeWitt and J. Naughton, "Dynamic Memory Hybrid Hash Join," Tech. Rep., University of Wisconsin, 1995.

[3] M. Nakayama, M. Kitsuregawa, and M. Takagi, "Hash-partitioned join method using dynamic destaging strategy," in *VLDB*, 1988, pp. 468–478.

[4] C. Walton, A. Dale, and R. Jenevein, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins," in *VLDB*, 1991, pp. 537–548.

[5] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *VLDB*, 1992, pp. 27–40.

[6] M. Kitsuregawa, M. Nakayama, and M. Takagi, "The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method," in *VLDB*, 1989, pp. 257–266.

[7] Y. Ioannidis, "The history of histograms (abridged)," in *VLDB*, 2003, pp. 19–30.

[8] S. Chaudhuri and V. Narasayya, "TPC-D data generation with skew," Tech. Rep., Microsoft Research, Available at: *ftp.research.microsoft.com/users/viveknar/tpcdskew*.