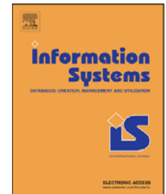
Information Systems 34 (2009) 493–510

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys



Using intrinsic data skew to improve hash join performance

Bryce Cutt, Ramon Lawrence *

Department of Computer Science, University of British Columbia, Okanagan, 3333 University Way Kelowna, British Columbia, Canada V1V 1V7

ARTICLE INFO

Article history:
Received 1 April 2008
Received in revised form
14 October 2008
Accepted 4 February 2009
Recommended by: O'Neil

Keywords:
Hybrid hash join
Skew
Histogram
Partition
Distribution

ABSTRACT

Hash join is used to join large, unordered relations and operates 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 the performance of hash join. 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 10–60% than hybrid hash join.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Hash join is the standard join used in database systems to process large join queries. Any performance improvement for hash join is significant due to the cost and prevalence of hash-based joins, especially in the large queries present in data warehouses and decision-support systems.

Hash join divides the two input relations into partitions using a hash function. The smaller relation is called the build relation and is partitioned first. The larger relation is called the probe relation. Hash join does not adapt its operation in response to skew in the join relations. Real data sets often contain skew. *Skew* occurs in data when certain values occur more frequently than others. 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.). Prior work has focused on how to maximize the performance of hash join by using memory efficiently and avoiding the negative effects of skew. Hybrid hash join (HHJ) [4] keeps one build partition in memory to reduce the number of I/O operations performed. Dynamic hash join (DHJ) [5,13] allocates more partitions than needed to subdivide the relation in order to counter the effects of skew. Build partitions are dynamically de-staged as memory is required. Avoiding partition skew has also been considered in distributed databases [7]. However, there has been no work that detects data skew in the probe relation in order to maximize the number of in-memory results produced.

In this paper, we present a modification to hash join that exploits data skew to improve hash join performance. The basic idea is to buffer in memory the tuples of the build relation whose join values occur most frequently in the probe relation. Keeping tuples in memory that have a higher probability of joining with other tuples reduces the number of I/Os performed. Although the general approach is straightforward, the complexity arises in handling situations of memory overflow and dealing with inaccurate estimates. The goal is to develop a stable algorithm that on average outperforms DHJ for skewed data sets and does not perform significantly worse when no skew is present or when the skew is estimated incorrectly.

* Corresponding author. Tel.: +1 250 807 9390.
E-mail addresses: brycec@interchange.ubc.ca (B. Cutt), ramon.lawrence@ubc.ca (R. Lawrence).

0306-4379/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2009.02.003

Our algorithm implementation uses histograms to detect skew in the input relations. Histograms [9] are commonly produced by a database system for query optimization and can be exploited at no cost by the join algorithm. Our *histojoin* algorithm has better performance than DHJ for skewed data.

The contributions of this paper are:

- An analysis of the advantage of exploiting data skew to improve hash join performance.
- A modification of DHJ 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. In Section 5 is a discussion of how to use histojoin in the presence of selections, queries with multiple joins, and histogram inaccuracies. Experimental results are in Section 6, and Section 7 has 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. The underlined attributes are the primary key attributes of the relations. Assume that the number of tuples of R, denoted as $|R|$, is smaller than the number of tuples of S (i.e. $|R|<|S|$). Assume $S.A$ is a foreign key to $R.A$.

HHJ [4] is a standard hash-based join algorithm in database systems. HHJ 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. HHJ uses any additional memory beyond what is needed for partitioning to store one partition in memory of the smaller (build) relation (R). While the larger (probe) 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 to disk tuples of the smaller relation and the matching tuples of the larger relation. Hash join completes its execution in a cleanup phase after the probe relation is partitioned. In the cleanup phase, each build partition on disk is loaded into a main memory hash table structure and probed with the corresponding probe partition. This is repeated for all on-disk partitions.

Most systems have no intelligent way of selecting which partition remains memory-resident. PostgreSQL simply selects the first partition. This assumption makes sense if the data set is uniform. In that case, each tuple in R is equally likely to join to tuples in S, so it does not matter what tuples in R are left in memory. If the data are 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.

DHJ [5,13] is similar to HHJ 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.

Skew can be classified [14] 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 [7] and DHJ [10,13]. Partition skew can be partially mitigated by using many more partitions than required, as in DHJ, and by producing histograms on the data when recursive partitioning is required. Handling partition skew is orthogonal to this work.

Intrinsic data skew is when data values are not distributed uniformly. Intrinsic data skew may cause partition skew for hash joins when the join attribute on the build relation is not the key attribute. Data skew causes values to occur with different frequencies in the relations. In our example that joins $R.A = S.A$ (primary-to-foreign key join), data skew may cause the distribution of values of $S.A$ (probe relation) to vary dramatically.

Histograms have been previously used during recursive partitioning [8] to detect data skew and minimize partition skew. However, no previous work has discussed using existing histograms in the database system to estimate and exploit skew during the first partitioning step.

Our histojoin algorithm is designed to use histograms as currently implemented in the database system. The algorithm does not assume any histogram method and will work with any method. Commercial systems typically implement equi-depth [12] or maxdiff (Microsoft SQL server) histograms. An overview of histograms can be found in [9]. The actual construction of the histograms is orthogonal to this work.

In the paper, we will use the TPC-H database benchmark standard as an example database. TPC-H is a decision-support and data warehouse benchmark developed by the Transaction Processing Performance Council (TPC). More information about the TPC-H benchmark can be found at [1]. The TPC-H schema diagram is in Fig. 1. The SF in the diagram represents the scale factor of the relations. We will use scale factors 1 and 10 which produce total database sizes of approximately 1 and 10 GB, respectively. The largest relation, *LineItem*, has just over 6 million records for SF = 1 and 60 million records for SF = 10.

3. General approach

The general approach is to use the extra memory available to the hash join to buffer the tuples that

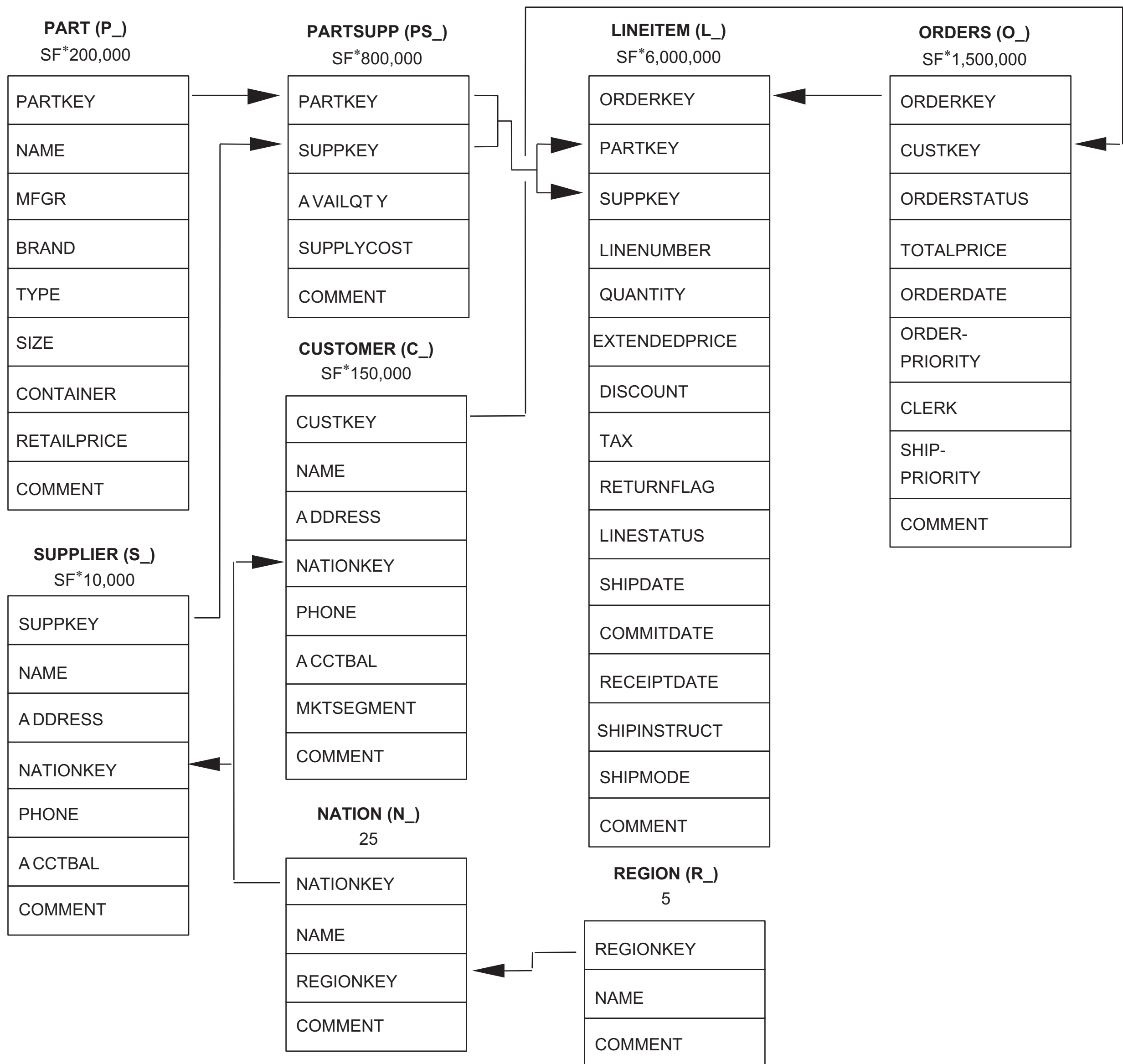


Fig. 1. TPC-H schema from [1].

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 HHJ 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. HHJ 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 values of $R.A$ are selected to be memory-resident. Tuples that do not fall into the ranges are partitioned using a hash function as usual. The data structures used are shown in Fig. 2.

In Fig. 2 there are three in-memory partitions (a, b, c) and 10 hash partitions numbered 0–9. For Level 1 partitions, each partition is defined by one or more join attribute value ranges. For example, partition a consists of values from 0 to 39 and 740 to 799. Ideally, these attribute

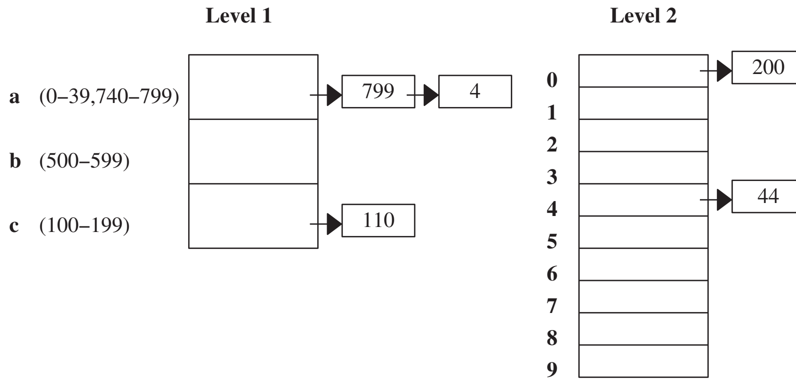


Fig. 2. Two level partitioning.

values are the most frequently occurring in S. The maximum partition size is bounded by the memory size available to the join. The Level 2 partitions are regular hash partitions. If a tuple does not fall into any of the Level 1 partitions, it is placed in a Level 2 partition by hashing the join attribute value. In general, there may be multiple Level 1 memory-resident partitions each defined by multiple ranges of values. The only constraints are that each partition must fit in the available memory during a cleanup phase at the end of the join, and the total memory used by in-memory partitions is always below the memory available.

3.1. Theoretical performance analysis

In this section we provide the theoretical maximum improvement of skew-aware partitioning using data distributions versus random partitioning (HHJ). Let f represent the fraction of the smaller relation (R) that is memory resident: $f = M/|R|$ (approximately), where M is the memory size. The number of tuple I/O operations performed by HHJ is $2 * (1 - f) * (|R| + |S|)$. The factor 2 represents the two I/Os performed for each non-memory-resident tuple: one to flush to disk if not memory-resident and then one to read again during the cleanup phase of the join. Note that this does not count the cost to read the tuple initially.

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 $2 * (1 - f) * |R| + 2 * (1 - g) * |S|$. The absolute difference in I/Os performed between histojoin and DHJ is $2 * (1 - f) * (|R| + |S|) - (2 * (1 - f) * |R| + 2 * (1 - g) * |S|)$ which simplifies to $2 * (g - f) * |S|$. The percentage difference in I/Os is $((g - f) * |S|)/((1 - f) * (|R| + |S|))$.

The absolute difference in total I/Os performed given selected values of f and g is given in Fig. 3. The percentage difference in total I/Os performed is given in Fig. 4. The absolute difference is directly proportional to the difference between f and g. The table shown is for a 1:1 ratio of R and S where $|R| = |S| = 1000$. The difference between f and g is bounded above by the intrinsic skew in the data set and is limited by how we exploit that skew during partitioning.

	g						
f	**5%**	**10%**	**20%**	**50%**	**80%**	**90%**	**100%**
5%	0	100	300	900	1500	1700	1900
10%	-100	0	200	800	1400	1600	1800
20%	-300	-200	0	600	1200	1400	1600
50%	-900	-800	-600	0	600	800	1000
80%	-1500	-1400	-1200	-600	0	200	400
90%	-1700	-1600	-1400	-800	-200	0	200

Fig. 3. Absolute reduction in total I/Os of skew-aware partitioning versus random partitioning for various values of f and g and $|R| = |S| = 1000$.

Properly exploiting data skew allows $g > f$, but if the in-memory tuples are chosen poorly, it is possible for $g < f$. This is worse than the theoretical average of the uniform case for HHJ. Tuples may be chosen improperly if the statistics used for deciding which tuples to buffer in memory are incorrect causing the algorithm to buffer worse than average tuples.

As an example, consider the "80/20 rule" for this data set. If we can keep 20% of tuples of R in memory ($f = 20\%$) that join with 80% of the tuples in S ($g = 80\%$), then skew-aware join will perform 68% fewer tuple I/Os than HHJ. However, if the data had "80/20 skew", but the 20% of tuples of R buffered in memory were the least frequently occurring in S, then it may be possible for $g = 5\%$, resulting in skew-aware join performing 17% more I/Os than hash join.

4. Histojoin algorithm

A low cost technique for performing skew-aware partitioning is by using histograms. Histograms [9] are used in all commercial databases for query optimization and provide an approximation of the data distribution of

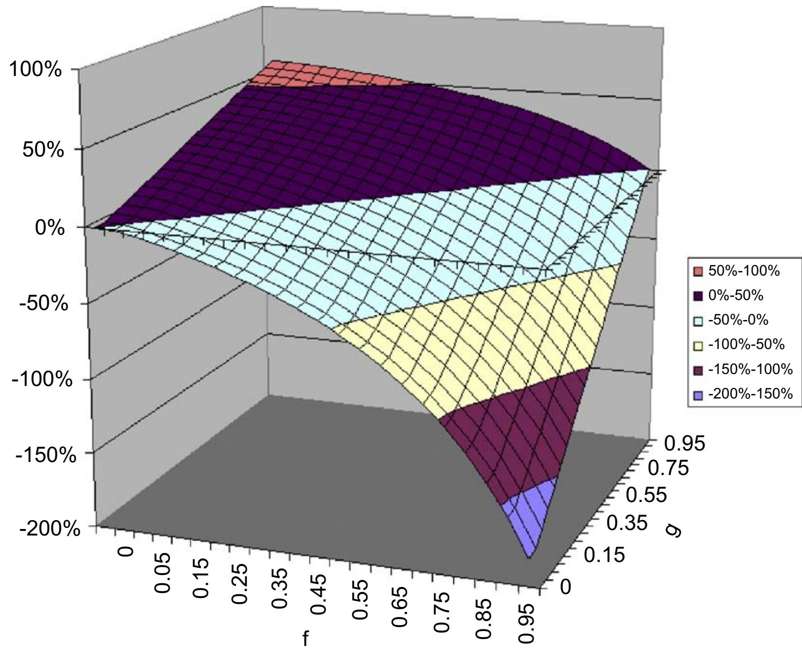


Fig. 4. Total I/Os percent difference.

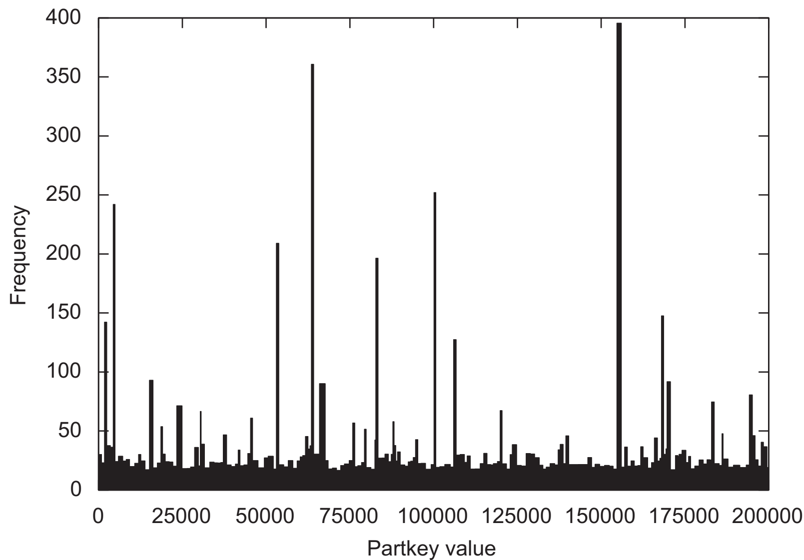


Fig. 5. Partkey histogram for *LineItem* relation TPC-H 1 GB Zipf distribution ($z = 1$).

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 Fig. 5.

The advantage of using histograms is that they are readily available, 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. An imprecise or out-of-date histogram limits histojoin's ability to exploit the data skew.

4.1. Algorithm overview

The histojoin algorithm works by implementing a set of privileged partitions in addition to the partitions DHJ would normally use. These privileged partitions are the last partitions to be flushed from memory to disk and are arranged so that they are flushed in a specific order,

whereas the non-privileged partitions are flushed in a randomized order. The privileged partitions correspond to the Level 1 partitions shown in Fig. 2.

The differences between histojoin and DHJ are isolated in the hash table. Histojoin's hash table is a two layered table that is aware of which partitions are privileged, how to determine if tuples fall in the privileged partitions, and how to optimally flush the partitions by first randomly flushing non-privileged partitions and then flushing the privileged partitions in order of worst to best.

The key difference between histojoin and DHJ is that histojoin attempts to isolate frequently occurring tuples in the privileged partitions which are the last ones flushed. DHJ spreads out frequently occurring tuples across all partitions and provides no special handling for frequent tuples.

A flowchart describing the histojoin algorithm is in Fig. 6. The first step is to load the histogram and determine which tuple value ranges are in the privileged partitions. At this point, if insufficient skew is detected or there is limited confidence in the histogram, a decision is made on the maximum size of the privileged partitions. If no skew is detected, histojoin will allocate no memory to the privileged partitions, and the algorithm behaves identically to DHJ. Determining the privileged partitions is discussed in Section 4.2.

Given sufficient detected skew, the privileged partition ranges are organized into efficient data structures to allow rapid determination of privileged tuples. Each build and probe tuple requires a range check to determine if they belong in a privileged partition. The range check operation is discussed in Section 4.3.

Histojoin processes the join in a similar manner to DHJ. Tuples are read from the build relation. When a tuple is read, the range check is performed. If the tuple falls into a privileged partition, it is placed there. Otherwise, the tuple's partition is determined using a hash function similar to DHJ. Whenever memory is full while the build relation is being read, a partition flush is performed. Non-privileged partitions are flushed first. If all non-privileged partitions are flushed, the privileged partitions are flushed in reverse order of benefit. When flushing the non-privileged partitions, we flush in random order. Once a partition is flushed, a single disk buffer is allocated to the partition to make writing tuples to the disk file more efficient. A flushed partition cannot receive any new tuples in memory. This has been referred to as a *frozen* [5,11] partition in previous work.

Once the build relation is completely read, there will be some build partitions in memory and others in disk files. Partitions that are memory-resident have main memory (chained) hash tables constructed to store their tuples. These hash tables will be probed using tuples from the probe relation.

The probe relation tuples are then read. The range check is performed on each tuple. If the tuple corresponds to an in-memory build partition (privileged or not), it probes the chained hash table for that partition to potentially generate results. If the corresponding build partition is not in memory, the probe tuple is written to the probe partition file on disk. Once all probe relation tuples are read, there will be pairs of build and probe partitions on-disk. Main memory is cleared, and each partition pair is read from disk and processed. Typically, the build relation partition is read, a chained hash table produced, and then results are generated by probing using probe relation tuples. However, common practices like

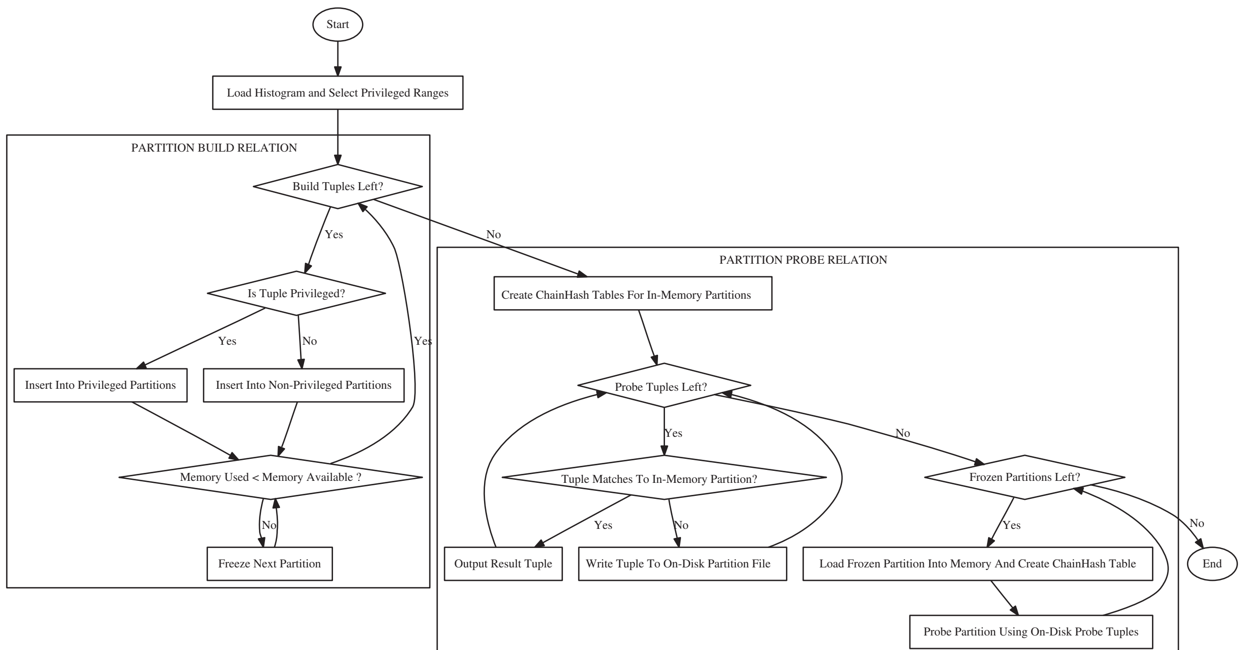


Fig. 6. Histojoin flowchart.

processing multiple partitions at the same time during this cleanup phase and role reversal can be applied.

In summary, histojoin behaves like DHJ except that its hash table structure allows for the identification and prioritization of frequently occurring tuples in the probe relation. The differences between DHJ and histojoin are embedded in the distribution of tuples between the two layers of the hash table and the order in which partitions are frozen to disk to free up memory. All other hash join techniques including bit vector filters, role reversal, etc. are unaffected by these modifications.

4.2. 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 in the privileged partitions. For instance in Fig. 5 there are several ranges of part keys that occur frequently in *LineItem*. The challenge is that the join partition size is determined independently from histogram partitioning. For example, let $|R| = 1000$ and $M = 100$. Thus, at least 10 partitions of R are required. The in-memory partition can have 100 tuples. It may require multiple independent ranges in the histogram to define a set of attribute ranges that contain up to 100 distinct values of R and have high frequency in S.

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 detailed steps are:

- Assume each histogram bucket entry is a 6-tuple of the form (*MINVAL*, *MAXVAL*, *ROWCOUNT*, *EQROWS*, *DISTINCT_ROWS*, *ROWS_R*). *MINVAL* and *MAXVAL* are the lower and upper (inclusive) values defining the bucket range. *ROWCOUNT* is the number of rows in S with a value in the range of $[MINVAL, MAXVAL)$. *EQROWS* is the number of rows in S whose value is exactly equal to *MAXVAL*. *DISTINCT_ROWS* is the distinct number of values in S in the bucket range. *ROWS_R* is a derived value (not present in the histogram) that is the estimate of the number of rows in R that have a value in the histogram bucket range. The estimation of *ROWS_R* is given in Section 4.2.1.
- A bucket *frequency* is calculated as

  $(ROWCOUNT + EQROWS)/ROWS_R.$
- 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

  $count + ROWS_R <= M.$
- 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 Fig. 7, and a join memory size of 400 tuples. The first histogram bucket added has range 751–1000 (250 tuples) as its frequency is 4.6. The second histogram bucket added has range 101–200 with frequency 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 to 100 can be divided into a subrange of 1–50.

Not all histograms will separate out the frequency of a boundary value (such as *EQROWS*) in that case the frequency is calculated as $ROWCOUNT/ROWS_R$. When a histogram does separate out the frequency of a boundary value (such as with maxdiff histograms), these values can be used as separate bucket ranges as they typically have very high frequencies. These single, high-frequency values are referred to as *premium values*. Premium values have a high payoff as they occupy little memory (one tuple each) and match with many rows in the probe relation. Premium values tend to be good values to keep in memory even when the accuracy in the histogram is low, especially when they are significantly more common than the average value.

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. Tuple with key 350 should be memory resident. Our algorithm creates separate one value ranges for each separation value. When sorted, these

MINVAL	MAXVAL	ROWCOUNT	EQROWS	DISTINCT_ROWS	ROWS_R	FREQ
1	100	300	5	100	100	3.05
101	200	300	10	100	100	3.1
201	350	150	100	150	150	1.67
351	500	200	40	150	150	1.6
501	750	244	6	250	250	1
751	1000	650	500	250	250	4.6

Fig. 7. Histogram partitioning example.

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, 750, 100, 751–999, 101–199, and 1–46. (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.

4.2.1. *Estimating cardinality of value ranges*

A histogram *estimates* the number of distinct values and number of tuples in a histogram bucket (*DISTINCT_ROWS* and *ROWCOUNT*, respectively). If the histogram is on the probe relation *S*, then the histogram provides the number of tuples in each bucket for relation *S*. However, histojoin also requires an estimate of the number of tuples in *R* that have a value in a histogram bucket. This estimate is used to determine approximately how many histogram buckets can be memory-resident for the build relation *R*. This value is also used to determine the relative value of each bucket. Histogram buckets with few rows in *R* and numerous rows in *S* are prime candidates for privileged partitions.

Given the number of distinct values in *S*, *DISTINCT_ROWS*, the estimate of the number of rows in *R* with values in that range, *ROWS_R*, is determined as follows:

- For integer values, it is calculated using the bucket low and high range values. That is, $ROWS_R = MAXVAL - MINVAL + 1$.
- For non-integer values, it is estimated as $ROWS_R = DISTINCT_ROWS$.

There will be inaccuracy in estimating *ROWS_R* for non-integer values. For one-to-many joins, *ROWS_R* will be underestimated due to primary key values not appearing in the foreign key relation. For many-to-many joins, it is impossible to determine exactly how many rows in *R* will have values in the range without having a histogram on *R* as well. Some heuristics can be used based on the size of relation *R*, but in general, the estimate may be imprecise. Thus, it is critical that histojoin adapts its memory management to flush even privileged partitions in case of inaccurate estimates. This is discussed further in Section 4.3.

4.3. *Partitioning*

Histojoin partitions tuples in two layers. The first layer contains privileged partitions with join attribute ranges that are defined as described in Section 4.2. A tuple is placed in a privileged partition if its join attribute value falls into one of the privileged partition ranges. This is performed using a range check function. A *range check* is performed for each tuple by comparing its join attribute value with the ranges calculated in Section 4.2. In our example, the ranges are 1000, 350, 500, 200, 750, 100, 751–999, 101–199, 1–46.

For efficiency, the range check is implemented in two steps. The first step uses a hash table to record all ranges of size one. Each hash table entry maps the join attribute to a partition number. This step is used for very frequently occurring values such as premium values. The size of this hash table is very small, usually less than 200 entries, as the number of premium values is limited based on the number of histogram buckets. The second step processes all ranges of size greater than one by storing them in a sorted array. This sorted array is searched using a binary search to detect if a value is in one of the ranges.

When a range check is performed the value is first tested against the hash table. If it is in the hash table then the mapped partition is returned. If not then a binary search is performed on the sorted array, and if the correct range is found the related partition is returned. If the value is not found in either of these two structures then it does not fall in a privileged partition, and the value is hashed to find which non-privileged partition it belongs in. For tuples with join values that do not fall into privileged partition ranges, the tuples are placed in hash partitions using a hash function. This hash partitioning works exactly the same as in DHJ.

This hash table and search array method works for all types and combinations of values. A further speed optimization for integer values is to enter every value that falls in a range into the hash table and not use the binary search. This works for integer values because the possible values in a range can be discretely enumerated.

5. Using histojoin

In this section, we discuss some of the issues in using histojoin. These issues include handling different join cardinalities, tolerating inaccuracy in histograms, and supporting joins where the input relations are from selection or other join operators.

5.1. *Join cardinality*

Although the previous examples considered primary-to-foreign key joins, histojoin works for all join cardinalities. Histojoin is useful when there is a histogram on the join attribute of the probe relation, and the probe relation has skew. If due to filtering the foreign key relation is the smaller (build) relation, histojoin is not usable because the probe (primary key) relation is uniform, and there is no skew to exploit. However, it may be possible to reverse the roles and still make the larger foreign key relation the probe relation if there is skew to exploit that improves performance.

Histojoin adds no benefit over DHJ for one-to-one joins due to the uniform distribution of the probe relation. In this case, histojoin behaves exactly as DHJ and allocates no privileged partitions.

For many-to-many joins, histojoin only requires the histogram on the probe relation. The algorithm behaves exactly as in the one-to-many case, but execution of the algorithm may result in flushing privileged partitions as the size estimates of the privileged build partitions are less accurate. For example, a histogram may indicate that the values from 5 to 10 have high frequency in the probe relation. Histojoin will estimate that there are six tuples in

the build relation in that range. However, there may be multiple occurrences of each value such that there are actually 30 tuples in the build relation with values in that range. This may force histojoin to flush some privileged partitions to compensate for the over-allocation of memory. A histogram on the build relation may mitigate some of these estimation concerns, but may be hard to exploit as independently produced histograms may have widely differing bucket ranges. Even when histojoin over-allocates privileged partition ranges, dynamic flushing based on frequency improves performance over DHJ while avoiding memory overflows.

The join cardinality cases are enumerated in Fig. 8.

5.2. Histogram inaccuracies

In the ideal case, the join algorithm would know the exact distribution of the probe relation and be able to determine exactly the skew and the frequently occurring values. Without pre-sampling the inputs, this requires a pre-existing summary of the distribution as provided by histograms. Histograms are not perfect because they summarize the information, which results in lack of precision. Also, the histogram may be inaccurate as it may be constructed by only using a sample of the data or was constructed before some modifications occurred on the table.

Note that skew-aware partitioning, as implemented by histojoin, can be used with a sampling approach as well as with pre-defined histograms. The advantage of using histograms is that there is no overhead during join processing as compared to sampling. The disadvantage is the accuracy of the distribution estimation may be lower. Histograms are valuable because they require no pre-processing for the join as they are pre-existing and are kept reasonably up-to-date for other purposes by the optimizer. We have experimented with non-random sampling by examining the first few thousand tuples of the probe relation before processing the build relation. Although it is sometimes possible to determine very frequent values using this approach, in general, most relational operators produce a set of initial tuples that is far from a random sample. True random sampling incurs cost that is too high for the potential benefit.

There are two key histogram issues. First, the histogram may not be a precise summary of a base relation distribution due to issues in its construction and maintenance in the presence of updates. Second, if the join is performed on relations that are derived from other relational operators (selection, other joins), then a histogram on the base relation may poorly reflect the distribution of the derived relation. Without an approach to derive histograms through relational operators, we must decide on our confidence in the histogram when allocating memory in the operator.

Without rebuilding or sampling to improve the accuracy of the histogram, which in general would increase the cost of the join operator, our approach assigns a *confidence value* to the histogram. The confidence value reflects the confidence in the accuracy of the histogram in relation to the data it is designed to represent. Histograms derived after selections have lower confidence than those recently built on the base relation.

The confidence value is used to determine how many privileged partitions are used. With a high confidence value, privileged partition ranges are defined such that almost all of the memory is allocated to the privileged partitions, as we are reasonably certain that the best tuples in the histogram are actually the best tuples in the relation. For a low confidence value, only the absolute best values as determined by the histogram are used as the range partitions. The result is that we can control our benefit or penalty as compared to DHJ based on the confidence of the estimates. This improves the stability, robustness, and overall performance of the algorithm.

For example, consider $M = 1000$ (1000 tuples can fit into memory). Let the join attribute value range be 1–2000. With a high confidence histogram, the algorithm would define privileged partition ranges to occupy all 1000 tuples of memory available. For instance, it may allocate four ranges 100–199, 300–599, 1000–1199, and 1500–1899 that would correspond to 1000 tuples in the build relation. With a low confidence histogram, the algorithm only allocates the very best ranges, which may result in only two ranges such as 100–199 and 1000–1199 (300 total tuples). Our algorithm determines the ranges to allocate based on the frequency of occurrence and the confidence value. With the low confidence histogram, the range 100–199 must have been significantly more common than average. The number of privileged partitions is reduced with a low confidence histogram to reduce the penalty of error. For instance, the range of values 100–199 may turn out to be very infrequently occurring in the probe relation. Buffering build tuples in the range 100–199 then would produce fewer results than buffering random tuples.

Type	Larger Side	Approach	Special Notes
1-1	Either	behave like DHJ	No skew in relations.
1-M	1	behave like DHJ	No skew in probe.
			Evaluate role reversal if skew on many-side.
1-M	M	use probe histogram	Skew can be exploited.
M-N	M or N	use probe histogram	Skew can be exploited.

Fig. 8. Join cardinality cases.

There are multiple possibilities for determining how many tuples to put in the privileged partition based on the histogram confidence level. One approach is to select ranges whose frequencies are one or more standard deviations better than the mean frequency of all ranges. The amount that the ranges must be better than the mean is increased for lower confidence histograms. A high confidence histogram will fill up memory with histogram buckets that are above the mean. A low confidence histogram will only accept buckets that are multiple standard deviations better than the mean.

The approach chosen to measure the quality of the histogram depends on the database system and its optimizer. Our two experimental implementations (see Section 6) use different approaches to selecting ranges based on the confidence level. The stand-alone Java implementation that only performs the joins and does not have an optimizer operates in two modes. Histograms on base relations with or without a selection operator are considered high confidence and all privileged ranges better than the average are selected. A low confidence histogram is when a base relation histogram is used to estimate the distribution of a relation produced by an intermediate join operator. In this case, only single premium values are used and no ranges. The PostgreSQL implementation exploits PostgreSQL's statistics that capture the most common values (MCVs) of an attribute. All MCVs are kept regardless of the histogram confidence and the equi-depth histogram is not used to select ranges. This is an effective approach as the penalty for being incorrect with MCVs is minimal, the payoff is potentially very high, and there is a high probability that MCVs of a base relation remain MCVs in derived relations. More details are in Section 6.

There are two potential "costs" in using this approach. The first is a *lost opportunity cost* that occurs when due to low confidence in the histogram we do not select ranges with frequently occurring values as privileged partitions. In this case, the performance of the algorithm could have been improved had it been more aggressive on selecting privileged partition ranges. However, the performance would be no worse than DHJ as any tuples that are not privileged get flushed randomly as in DHJ. The second cost, *inaccuracy cost*, is much more important. Inaccuracy cost occurs when a value range is selected as privileged and turns out to be less frequently occurring than average. For example, if the 100 build tuple values in the range 100–199 map to two tuples on average in the probe relation, and the average build tuple maps to three tuples on average, then skew-aware partitioning will have worse average performance than DHJ. For low confidence histograms, it is better to be conservative in selecting privileged ranges, as there is a penalty for being too aggressive. By selecting no privileged ranges, histojoin behaves exactly as DHJ.

5.2.1. Handling selections

The discussion so far has considered joins where both inputs are base relations. It is common that a selection is performed before a join. A selection on the probe relation may change the distribution of join values and result in lower confidence in the histogram. The confidence can be changed based on the attribute correlation. If the selection attributes are highly correlated with the join attribute, then the histogram will most likely be very inaccurate. If there is low correlation, then the histogram is more usable and the uniform assumption can be applied. For example, the uniform assumption assumes that if a selection reduces the cardinality of the entire relation by 90%, then the cardinality of each histogram bucket is also reduced by 90%. If present, multi-dimensional histograms on both the selection and join attributes may be used to estimate the distribution after selection. It is also possible to use SITs (Statistics on Intermediate Tables) [2] to more accurately estimate the distribution.

Selections on the build relation are less problematic. A selection on the build relation may affect the number of build tuples in a privileged partition range. For instance, if the algorithm determines that the range 100–199 is valuable, it expects 100 unique values in the build relation. However, a selection may cause the actual number of build tuples to be 50. This is another example of a lost opportunity cost because given this knowledge, the algorithm may have been able to select more privileged partitions (since memory is available) or select different ones because the value of the partition range may be lowered since not all of its build tuples participate in the join. Note that since we do not allocate a static amount of memory to privileged partitions, the extra memory for the 50 tuples is available for other partitions (most likely non-privileged hash partitions) to use. The algorithm will still outperform DHJ if the build tuples actually in the privileged partition range join with more probe tuples than the average build tuple.

5.2.2. Multiple join plans

When a query consists of multiple joins, histojoin can be used with each join as long as a histogram is available or can be estimated for the probe relations. Histojoin can be used for star joins which are very common in data warehouses.

For example, consider a star join of the tables *Part*, *Supplier*, and *LineItem* as shown in Fig. 9a. With histograms on *LineItem.partkey* and *LineItem.suppkey* and no selection operations, histojoin will have high confidence histograms for both joins. The bottom join of *LineItem* and *Supplier* will use the histogram on *LineItem.suppkey*. The second join will use the histogram on *LineItem.partkey* which will accurately reflect the distribution of *LineItem.partkey* in

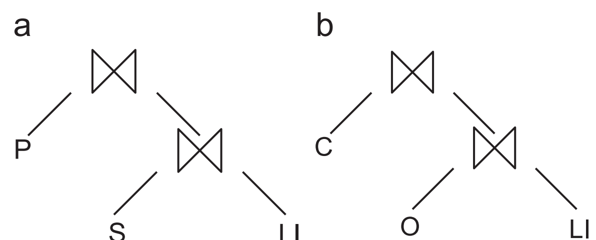


Fig. 9. Example multiple join plans. (a) *Part Supplier LineItem*; (b) *Customer Order LineItem*.

the intermediate join result *LineItem–Supplier* as the intermediate result was produced using a primary-to-foreign key join. In general, star joins with no selections and histograms on all join attributes of the fact table are accurately estimated and result in large performance improvements for skewed data.

In contrast, consider a join of the tables *Customer*, *Orders*, and *LineItem* as shown in Fig. 9b. The join of *LineItem* and *Orders* can exploit the histogram on *LineItem.orderkey*. However, the top join has no base histogram that can model the distribution of *custkey* in the intermediate relation *LineItem-Orders*. It is possible to estimate the histogram from one on *Orders.custkey*, but it would be a low confidence histogram. When selections are added with joins, the confidence of the histograms decreases, especially with selections on the probe relations.

5.3. Query optimizer modifications

There are minimal query optimizer modifications required to use histojoin. Histojoin can be used as a drop-in replacement to hybrid or DHJ, or the concepts used to modify an existing hash join implementation. If histojoin is implemented separately from HHJ, then when costing a potential join, histojoin will return a high, not-applicable cost for joins where a histogram does not exist or cannot be estimated for the probe relation. As a drop in replacement for HHJ the cost for histojoin when it cannot make use of histograms would be exactly equal to that of HHJ. The cost of histojoin will be the same formula as given in Section 3.1. In estimating the term g in the formula from the histogram we first calculate which histogram buckets will be in the privileged partitions. The histogram tells us how many probe tuples are related to each histogram bucket so using this we can estimate the number of results we will get from the in memory build tuples. In practice this can be done in one step as the estimate of result tuples can be calculated while choosing privileged build tuple ranges.

Given the list of privileged partitions, g is estimated by summing up the $FREQ * ROW_R$ or alternatively $ROWCOUNT + EQ_ROWS$ (see Section 4.2) then dividing by S. That is, $g = (\sum ROWCOUNT + EQ_ROWS)/|S|$.

Using the example histogram given in Fig. 7, without separating out the max values, the ranges in sorted order are 751–1000 ($W = 4.6$), 101–200 ($W = 3.1$), 1–100 ($W = 3.05$), 201–350 ($W = 1.67$), 351–500 ($W = 1.6$), and 501–750 ($W = 1$). In the example, the build relation R has 1000 tuples, and the probe relation S has 2505 tuples. With 350 tuples of memory ($f = 35\%$) the first two ranges 751–1000 and 101–200 would be selected as privileged and $E = 250 * 4.6 + 100 * 3.1 = 1460$. $g = E/|S| = 1460/2505 = 58\%$. Using the formulas in Section 3.1 we expect DHJ to perform 4556 I/Os during this join and histojoin to perform 3405 I/Os (25% less).

Using histojoin this way will allow a query optimizer to only use the algorithm when histojoin indicates that it will have a performance benefit (by exploiting the skew it potentially sees). A major benefit is that no major changes to the optimizer are required. The only issue is the DBMS must make the histograms available to the histojoin operator when costing and initializing.

Histojoin's performance and applicability are increased according to the database system support for statistics collection. For instance, histojoin works best when provided with a list of the most frequent values and their frequencies. It is this list of values and their associated tuples that must remain memory-resident. Some statistics systems collect this data explicitly either separate from the histogram (PostgreSQL's MCVs) or as part of the histogram (end-biased histograms). Note that histogram bucket ranges are a less accurate approximation to the most frequent value list.

The challenge of using histojoin on derived operators (selections, joins, etc.) can also be mitigated by better statistics collection. For example SITs [2] and statistics collection on views allow the optimizer to have improved distribution estimates for relations of intermediate operators. Instead of base histograms and uniform assumptions, these approaches can provide histojoin with more accurate data when deciding on privileged ranges/values. Any technique to improve the histogram accuracy will improve histojoin's performance. In summary, histojoin will always produce a correct result that is robust in the case of poor estimates and optimal according to the distribution estimate given. The more accurately the histogram reflects the actual data distribution, the better actual performance histojoin will have.

6. Experimental results

We present two separate experimental evaluations for histojoin. The first evaluation is a stand-alone Java application performing the joins. The second evaluation is an implementation of the algorithm in PostgreSQL. The histojoin algorithm was tested with the TPC-H data set. We used the TPC-H generator produced by Microsoft Research [3] 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 and 10 GB and labeled as skewed ($z = 1$) and high skew ($z = 2$).

6.1. Stand-alone evaluation

The dynamic version [5] of HHJ [4] (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 partitions using a histogram and DHJ flushed partitions to free memory without regard to the data distribution. DHJ typically flushes partitions in a deterministic ordering, but our implementation flushes randomly such that a more accurate average case is found. For instance, a deterministic ordering may always flush the exact worst partition for a join first. A random ordering will flush the worst

partition first with probability $1/P$ (where P is the number of partitions).

The data files were loaded into Microsoft SQL Server 2005, and histograms were 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 Athlon 64 3700+ (2.2 GHz) with 1.5 GB RAM running Windows XP Pro and Java 1.6. All results are the average of 10 runs. These results use TPC-H scale 1 GB and demonstrate the applicability of the algorithm in various scenarios.

6.1.1. Primary-to-foreign key joins

The joins tested were *LineItem–Part* on *partkey* and *LineItem–Supplier* on *suppkey* for $z = 1$ and 2. Memory fractions, *f*, were tested ranging from 10% to 100%.

Histojoin performs approximately 20% fewer I/O operations with the $z = 1$ data set which results in it being about 20% faster overall. This is a major improvement for a standard operation like hash join. An improvement occurs over all memory sizes until full memory is available for both joins (Fig. 10).

For the $z = 2$ data set, the performance difference is even more dramatic. In the 10% memory case histojoin performs 60% fewer I/Os resulting in 60% faster execution. The results by total I/Os and by time are in Figs. 11a and b, respectively.

DHJ 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

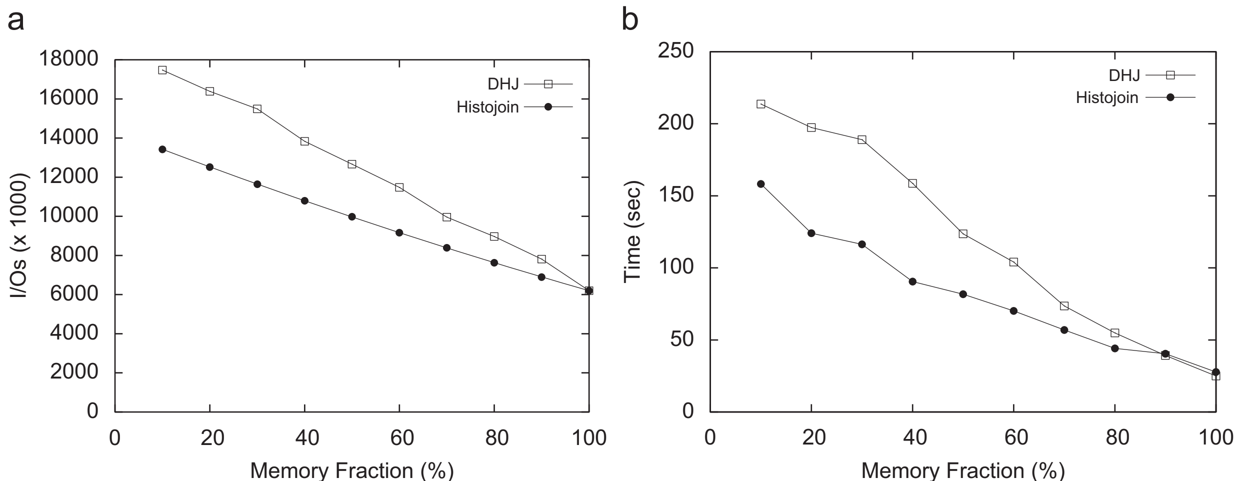


Fig. 10. *LineItem–Part* join ($z = 1$). (a) Total I/Os; (b) total time.

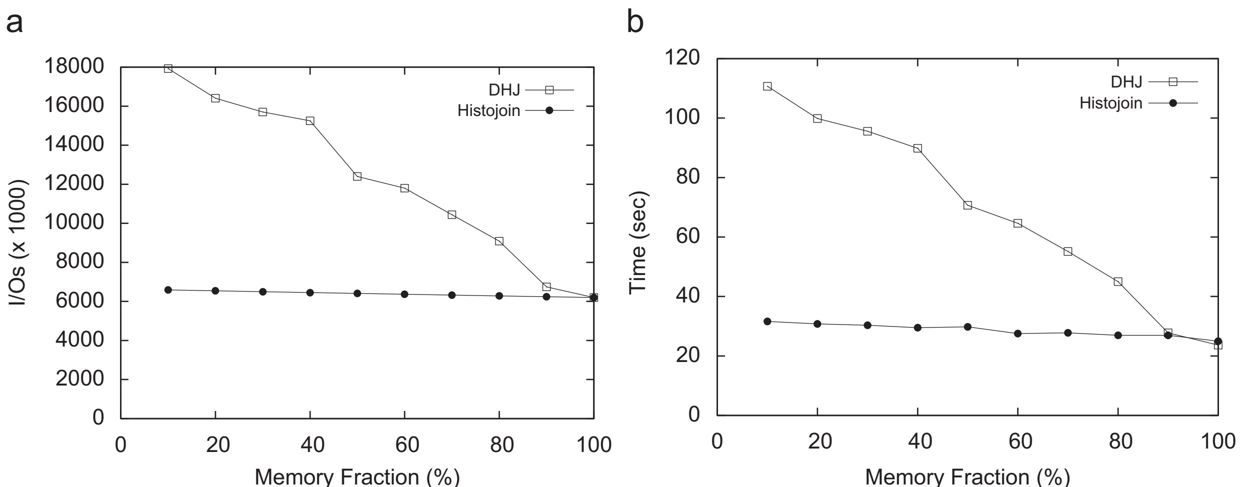


Fig. 11. *LineItem–Part* join ($z = 2$). (a) Total I/Os; (b) total time.

join has only a 10% probability of keeping the partition in memory with the key value that is most frequently occurring. The performance of DHJ is *unpredictable* for skewed relations and is highly dependent on the partition flushing policy. For highly skewed relations and low memory percentages the likelihood of DHJ flushing the best values is *very* high.

For the $z = 1$ data set and *LineItem–Supplier*, histojoin performs about 10–20% fewer total I/Os and executes 10–20% faster. For the $z = 2$ data set, histojoin performs between 20 and 60% fewer total I/Os and executes 20–60% faster. A summary of the percentage total I/O savings of histojoin versus DHJ for all joins is in Fig. 12.

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. For totally uniform data, histojoin selects no privileged partitions. For mostly uniform data, such as generated by the standard TPC-H generator, there are still some join attribute ranges that are marginally better and are used by histojoin to improve performance slightly.

6.1.2. Many-to-many joins

The many-to-many join tested combined a randomized version of the *Wisconsin* relation [6] with a randomized and Zipfian skewed ($z = 1$) version of the Wisconsin relation on the *tenPercent* column. Both relations contained 1,000,000 tuples. The *tenPercent* column has a domain that is 10% the size of the relation. For 1,000,000 tuple relations, the domain of *tenPercent* is 100,000. Memory fractions, *f*, were tested ranging from 10% to 100%.

For this test, the build relation (randomized *Wisconsin*) contains 1,000,000 tuples, and the *tenPercent* column contains values in the range 0–99,999, each value being shared by 10 tuples. The probe relation has a domain of 0–99,999 as well with a Zipfian distribution of the values. In the generated Zipfian relation, the top two values occur in 127,380 of the 1,000,000 tuples (12.7%) and 31,266 of the 1,000,000 tuples (3.7%), respectively. Beyond the top 200 values, the average value occurs in approximately 5.7 of the 1,000,000 tuples. A histogram on the probe relation column is misleading because it shows an integer domain of 100,000 tuples which underestimates the size of each privileged relation partition by a factor of 10.

This underestimation causes histojoin to allocate too much memory for privileged partitions because it thinks the partitions contain far fewer tuples than they really do. However, these privileged partitions are dynamically flushed as required with no harm to the performance. The *Wisconsin* results by total I/Os (includes cost of reading each relation) for this join are in Fig. 13. For all memory sizes, histojoin performs approximately 10% fewer IO operations than DHJ.

6.1.3. Histogram inaccuracies

To demonstrate the effect of histogram inaccuracies on join performance, a modified TPC-H *LineItem* relation was created to show the worst case scenario for histojoin and how the use of histogram confidence mitigates this scenario. The new *LineItem* relation contains only every 10,000th *partkey* $(1, 10,000, 20,000, \ldots, 200,000)$ and each of these values occurs as often as the others. A histogram was created that indicates to histojoin that these 10,000th values never occur in *LineItem* so that in all cases except the 100% memory case histojoin will not store any of the corresponding build tuples from the *Part* relation in memory but will instead fill memory with build tuples whose partkey values never occur within *LineItem*.

Histojoin was compared to DHJ using the join *LineItem–Part* on *partkey*. Memory fractions, *f*, were tested ranging from 10% to 100%. Histojoin executed the join under two confidence levels. In the high confidence level, it assumed the histogram was very accurate and fully allocated privileged partitions to memory. In Fig. 14, this corresponds to the HJ Bad histogram plot. Histojoin does considerably worse than DHJ by trusting a totally wrong histogram. In comparison, when executed under a low confidence level, histojoin only selects premium values from the histogram (in the diagram as HJ Bad Premium Values). Since the histogram is completely inaccurate, the premium values give no performance improvement, but also result in little cost compared to DHJ due to the minimal amount of memory occupied. If the histogram is

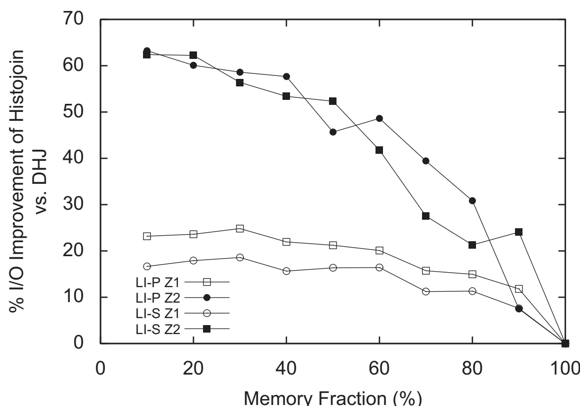


Fig. 12. Percentage improvement in total I/Os of histojoin vs. hash join.

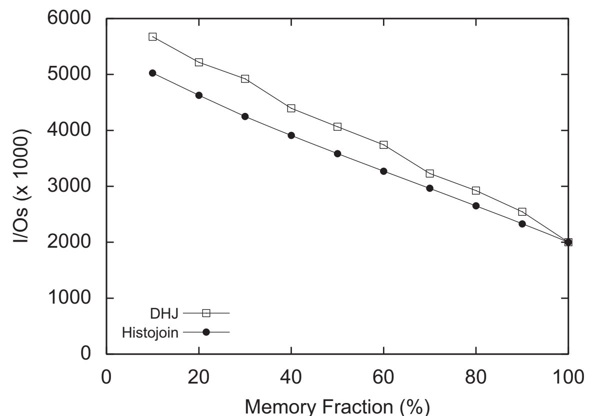


Fig. 13. Total I/Os for Wisconsin M–N Join ($z = 1$).

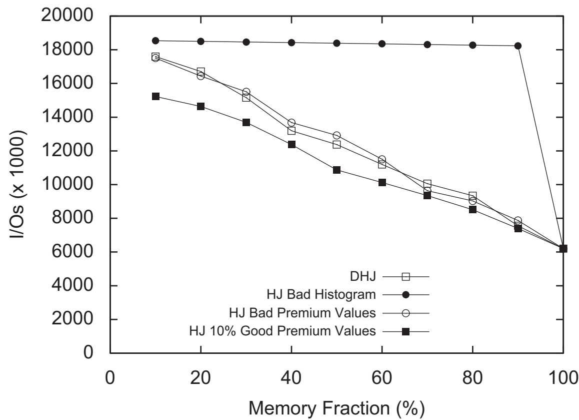


Fig. 14. Total I/Os for *LineItem–Part* join with histogram inaccuracies.

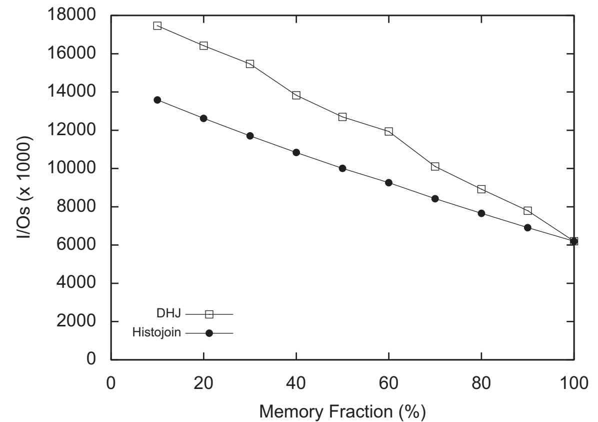


Fig. 15. Total I/Os for *LineItem–Supplier* join on string key ($z = 1$).

only 10% correct (10% of the premium values are good), histojoin in low confidence mode outperforms DHJ. In summary, executing histojoin in low confidence mode has little risk and considerable reward if the histogram is even marginally accurate.

6.1.4. Joins on string keys

With string keys histojoin is less accurate in predicting the size of build partition ranges for privileged partitions. To test joining on string keys, versions of the *LineItem* and *Supplier* relations were generated with the *suppkey* replaced by randomly generated strings. Once again memory fractions, f, were tested ranging from 10% to 100%. Much like the join of *LineItem–Part* on *partkey* using integer keys histojoin performs around 20% fewer Total I/Os than DHJ for the $z = 1$ data set. The results are in Fig. 15.

6.1.5. Multi-way joins

As described in Section 5.2.2, histojoin can be used on multi-way star joins when a histogram exists on the join attributes of the fact relation. A star join of the tables *Part*, *Supplier*, and *LineItem* as shown in Fig. 9a falls into this category.

If the memory for the entire query is split evenly between the two joins then for a memory percentage above 10% the first join of *LineItem–Supplier* would be done completely in memory as *Supplier* is quite small in comparison to *Part*. For this reason histojoin was compared to DHJ using memory fractions ranging from 3% to 10%. The total I/Os for the entire join using a $z = 1$ data set are shown in Fig. 16a. For all memory sizes histojoin performs about 20% fewer I/Os than DHJ. For memory sizes above 10%, histojoin is faster than DHJ but only one join requires disk I/Os. Results for the $z = 2$ data set are shown in Fig. 16b. Due to the high skew, histojoin dramatically improves on the performance of DHJ.

6.2. PostgreSQL implementation

Histojoin was implemented in PostgreSQL 8.4 to test its performance for large-scale data sets in a production system. PostgreSQL implements HHJ. Its HHJ implementation requires that the number of partitions be a power of two, and it always keeps the first partition in memory. Thus, our experimental data only collect data for memory fractions: 3.1% ($\frac{1}{32}$), 6.2% ($\frac{1}{16}$), 12.5% ($\frac{1}{8}$), 25% ($\frac{1}{4}$), 50% ($\frac{1}{2}$), and 100%.

PostgreSQL collects statistics on its tables. Statistics on an attribute of a table include the MCVs and an equi-depth histogram. The user is able to control on a per table basis the number of MCVs. The user can also initiate statistics re-calculations. The query optimizer has access to the histograms and a list of MCVs that are automatically generated for foreign key columns.

Histojoin was added to the PostgreSQL HHJ implementation. Using environment flags that PostgreSQL uses to control which joins are available, histojoin can be turned on and off from the standard HHJ implementation. Thus, we altered the existing HHJ implementation instead of implementing two hash join algorithms for the optimizer to choose between.

Histojoin requires the ability to use the existing statistics which were available in the planner. Our code used the join attributes of the probe relation to find statistics for that attribute. If no statistics were available, histojoin would not be used. If statistics are available, we only used the MCVs (not the histogram) as the MCVs are more precise. However, this means that the privileged partitions did not occupy very much of the memory available for build relation tuples. The MCVs were determined and allocated into an in-memory hash table when the join operator was initialized. The default number of MCVs is 10, so the size of the hash table is small (less than 1–4K). Our hash table size is at least four times the size of the number of MCVs (load factor is less than 25%) and collisions are resolved using open addressing.

During the partitioning of the build relation, a build tuple's join attributes are hashed according to the small MCV hash table to determine if its value is one of the MCVs. If it is, then the tuple is put into the hash table, otherwise it is processed using the regular hash function as usual. While partitioning the probe relation, a probe

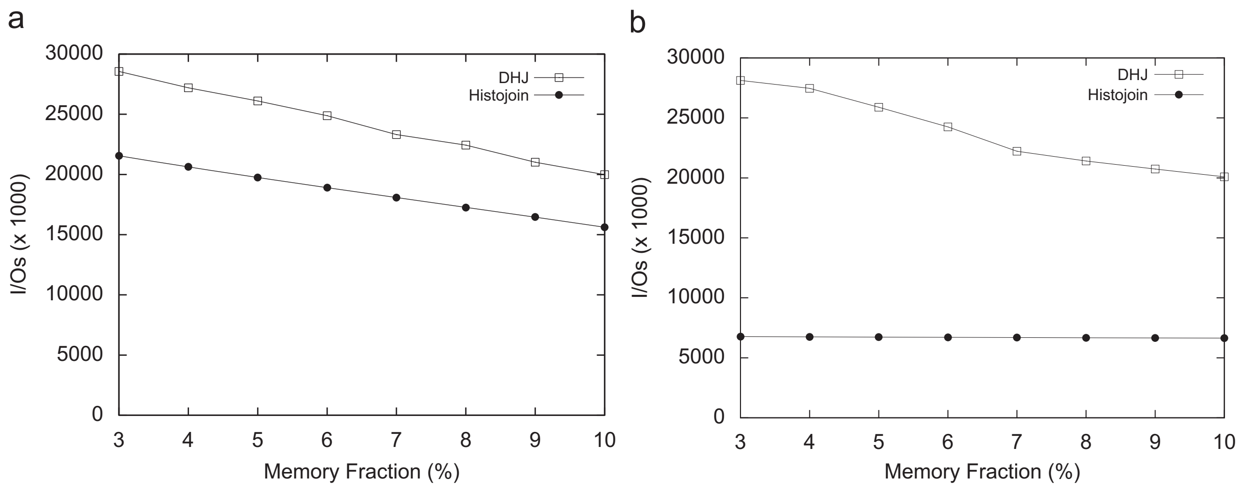


Fig. 16. Total I/Os for *LineItem–Supplier–Part* join. (a) $z = 1$; (b) $z = 2$.

tuple's join attribute is hashed and a lookup performed in the MCV table. If there is a match, the tuple is joined immediately, otherwise it proceeds through the hash join as normal. In effect, the MCV lookup table is a small mini-hash table for the most frequent values. Its size and cost would be less than or comparable to bit vector filtering.

The equi-depth histograms are not used as it is preferable to increase the number of MCVs rather than allocate ranges from the histogram. The experiments all use the default of 10 MCVs unless otherwise specified. Results are improved when MCVs are set to 100 or more.

The experimental machine for the PostgreSQL implementation was an Intel Core 2 Quad Q6600 (2.4 GHz) with 8 GB RAM running 64bit Debian Linux with a 2.6.25-2-amd6 kernel. These results use TPC-H scale 10 GB. Note that even for machines with large main memories, a join operator is allocated only a small fraction of the memory available as it must compete with other operators and other queries for system resources. The default join memory size for PostgreSQL is 1 MB. The experiments alter that memory size to produce the desired memory fraction based on the build table size.

There are a couple of differences from the Java experiments that should be noted. First, the execution times more accurately reflect the number of I/Os performed. This is due to increased stability and performance of PostgreSQL on Linux versus a Java implementation on Windows. The Java I/O counts are exact, but the execution times are more variable. There is less I/O performance improvement for the PostgreSQL implementation compared to the Java implementation because the PostgreSQL implementation only has very small privileged partitions (just the MCVs) where the Java implementation uses all available memory for privileged partitions by filling them with valuable histogram bucket ranges.

6.2.1. Primary-to-foreign key joins

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 Figs. 17a and b, respectively. Histojoin is around 10% faster and performs 10% less I/Os than HHJ. With the $z = 2$ data set, histojoin performs approximately 50% faster (Figs. 18a and b). The percentage improvement of histojoin is shown in Fig. 19. Note that the sudden improvement of HHJ for the $z = 2$ 50% memory case is because HHJ manages to get the best tuples from the build partition in its in-memory partition by chance.

6.2.2. Multi-way joins

When performing a star join of the tables *Part*, *Supplier*, and *LineItem* any memory size above 10% of the size of the *Part* table will run the smaller join of *LineItem* and *Supplier* completely in memory. This multi-way join was tested with memory fractions (sizes) of 0.78% (2770 KB), 1.56% (5440 KB), and 3.12% (10880KB). Fig. 20a shows that for the $z = 1$ data set Histojoin performs around 6% fewer IOs than HHJ and for the $z = 2$ data set Histojoin performs around 40% fewer IOs than HHJ.

6.2.3. Effect of number of MCVs

By increasing the number of MCVs from the default 10, the performance of histojoin increases as histojoin is able to capture more of the most valuable tuples. The join of *LineItem* and *Part* was performed with a memory size of 6.2% and various amounts of MCVs. The results by total I/Os and by time are in Figs. 21a and b, respectively. The query was run with 10, 100, 300, 500, 700, and 1000 MCVs on *partkey*. Histojoin's performance with the $z = 1$ data set can be increased by adding more MCV statistics as this data set has many relatively good MCVs. As more MCVs are added the benefit per new MCV is much less.

6.3. Results summary

For skewed data sets, histojoin dramatically outperforms DHJ by 10–60%. This is significant because DHJ is a very common operator used for processing the largest

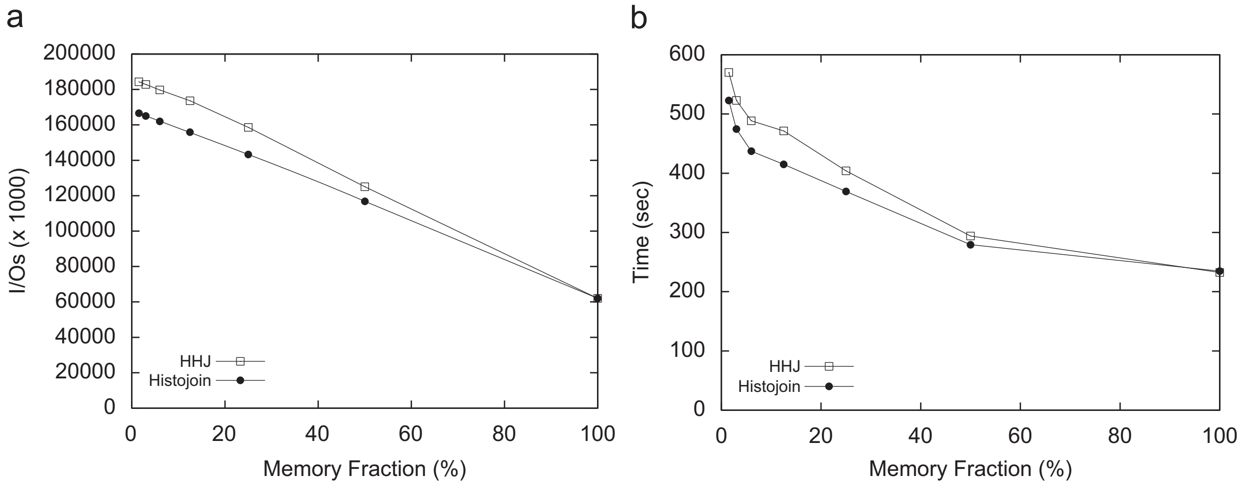


Fig. 17. 10 GB *LineItem–Part* join ($z = 1$). (a) Total I/Os; (b) total time.

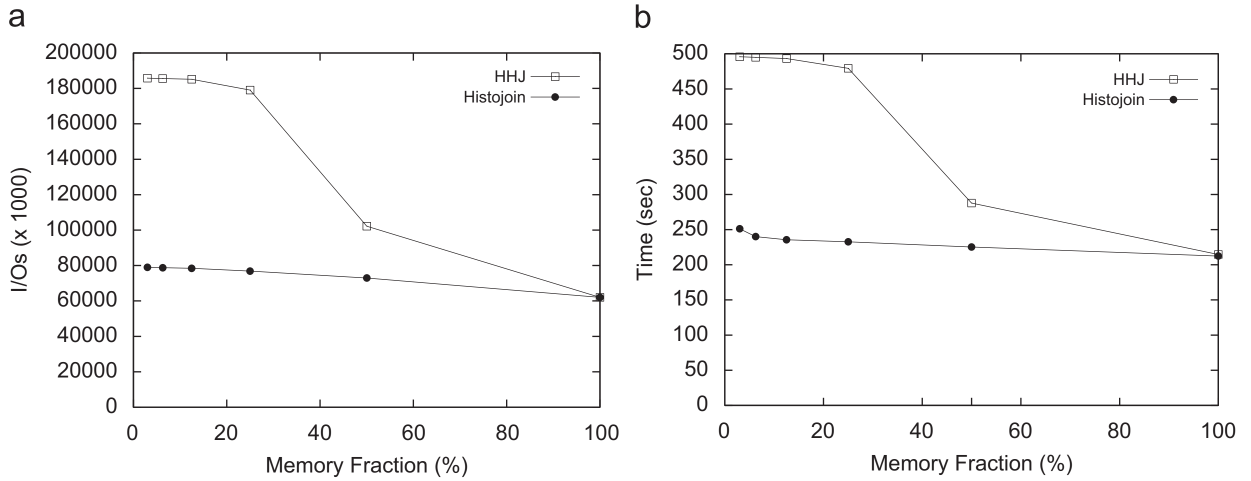


Fig. 18. Ten GB *LineItem–Part* join ($z = 2$). (a) Total I/Os; (b) total time.

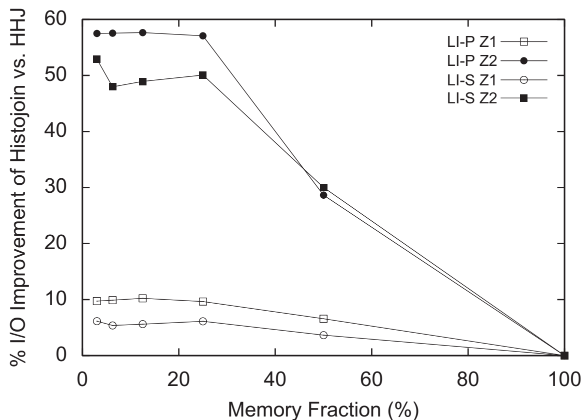


Fig. 19. Percentage improvement in total I/Os of histojoin vs. hash join (10 GB).

queries. As the amount of skew increases, the relative performance improvement of histojoin increases.

Histojoin introduces no performance penalty compared to DHJ 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). Histojoin has better performance with a more accurate estimate of the distribution of the probe relation. When the confidence in the histogram approximation of the distribution is low, histojoin allocates fewer privileged partitions which must be significantly better than the average. Thus, histojoin will exploit whatever skew is detectable and fall back to DHJ behavior otherwise. Even with low accuracy histograms, histojoin will improve join performance over hash join for skewed data sets.

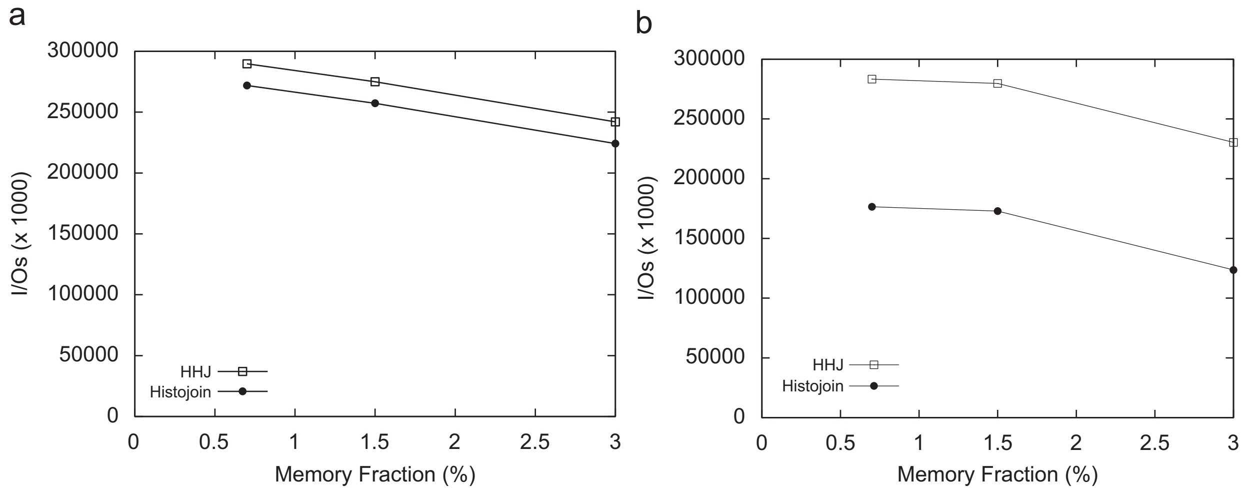


Fig. 20. Total I/Os for *LineItem–Supplier–Part* join (10 GB). (a) ($z = 1$); (b) ($z = 2$).

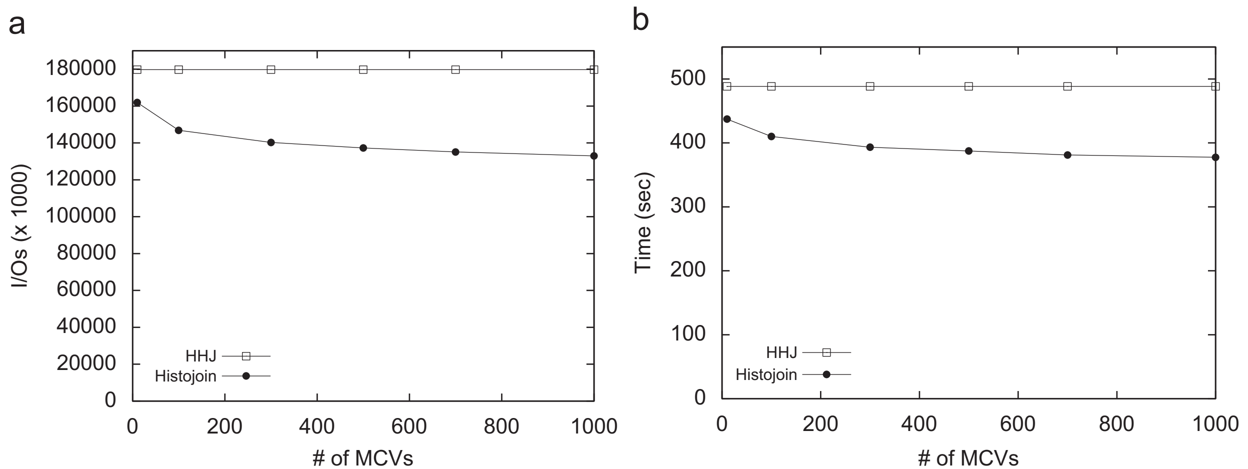


Fig. 21. Ten GB *LineItem–Part* join with various amounts of MCVs ($z = 1$). (a) Total IOs; (b) total time.

The implementation of histojoin in PostgreSQL uses only premium values determined from pre-generated MCV lists to determine its privileged partitions. Histojoin is minimally affected by bad estimates as the MCV lists are small and represent only a minimal memory overhead. In the experiments this implementation shows a large improvement over the standard HHJ operator used for all large unsorted joins in PostgreSQL while adding no noticeable overhead when skew cannot be exploited. Histojoin is especially valuable for smaller memory fractions as its relative benefit over HHJ is higher.

7. Conclusions

Intrinsic data skew is prominent in databases, and 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 10% to 60%) for skewed data.

Future work includes submitting the histojoin modifications to PostgreSQL and investigating how approaches like statistics on views can be used in conjunction with histojoin.

References

[1] TPC-H Benchmark, Technical report, Transaction Processing Performance Council, available at: ⟨http://www.tpc.org/tpch/⟩.
[2] N. Bruno, S. Chaudhuri, Exploiting statistics on query expressions for optimization, in: ACM SIGMOD, 2002, pp. 263–274.
[3] S. Chaudhuri, V. Narasayya, TPC-D data generation with skew, Technical Report, Microsoft Research, available at: ⟨ftp.research.microsoft.com/users/viveknar/tpcdskew⟩.
[4] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, D. Wood, Implementation techniques for main memory database systems, in: ACM SIGMOD, 1984, pp. 1–8.
[5] D. DeWitt, J. Naughton, Dynamic memory hybrid hash join, Technical Report, University of Wisconsin, 1995.

[6] D.J. DeWitt, The Wisconsin Benchmark: Past, Present, and Future, 1993.
[7] D.J. DeWitt, J.F. Naughton, D.A. Schneider, S. Seshadri, Practical skew handling in parallel joins, in: VLDB, 1992, pp. 27–40.
[8] G. Graefe, Five performance enhancements for hybrid hash join, Technical Report CU-CS-606-92, University of Colorado at Boulder, 1992.
[9] Y.E. Ioannidis, The history of histograms abridged, in: VLDB, 2003, pp. 19–30.
[10] M. Kitsuregawa, M. Nakayama, M. Takagi, The effect of bucket size tuning in the dynamic hybrid GRACE hash join method, in: VLDB, 1989, pp. 257–266.
[11] R. Lawrence, Early hash join: a configurable algorithm for the efficient and early production of join results, in: VLDB 2005, 2005, pp. 841–842.
[12] M. Muralikrishna, D.J. DeWitt, Equi-depth histograms for estimating selectivity factors for multi-dimensional queries, in: H. Boral, P.-Å. Larson (Eds.), ACM SIGMOD, ACM Press, New York, 1988, pp. 28–36.
[13] M. Nakayama, M. Kitsuregawa, M. Takagi, Hash-partitioned join method using dynamic destaging strategy, in: VLDB, 1988, pp. 468–478.
[14] C.B. Walton, A.G. Dale, R.M. Jenevein, A taxonomy and performance model of data skew effects in parallel joins, in: VLDB, 1991, pp. 537–548.