Accelerating Hash Join Performance by Exploiting Data Distribution

Yang Liu[#1], Zhen He[#2], Xiang Wu Meng [#3]

[#1] *Engineering Consulting Department, Sinopec Engineering Incorporation,*
21# Bldg.Anyuan,Anhuibeili,Chaoyang Dist.Beijing,China

[#2] *Department of Computer Science and Computer Engineering, La Trobe University*
Plenty Road, Bundoora, Melbourne Victoria, Australia

[#3] *School of Telecommunication Engineering, Beijing University of Posts and Telecommunications,*
10# Bldg.Xitucheng,,Haidian Dist.Beijing,China

Abstract**—***thejoin operator in relational databases is one of the most IO intensive operations. Thelarge size of input relations makes it hard to fit them entirely in RAM during join processing. Therefore therelations are processed in chucks inside a RAM buffer of limited size.The ideabehindasuccessfuljoin algorithm is to make the most efficient use of the limited sized buffer to minimizethenumberof IOs. The hash join algorithm has been a popular algorithm due to its relativelylowIOcostscompared to other methods. In this paper we make the observation that the performanceofthehash join can be dramatically improved if we take advantage of skewed distributionsandmissingvalues in join attributes. We propose the filtered hash join (FH-join) which filtersouttuplesoftheinput relations during the partitioning phase of the hash join to minimize the workleft forthejoinphase. The results show FH-join can outperform the hybrid hash join by up to a factor 4 in terms of total execution time when the data is much skewed.*

Keywords —*hash join, relational databases, query processing*

I. INTRODUCTION

It is commonly accepted that the join operator is often the most expensive operator when executing a database query. For example Hayes et al. [1] found the join operator on average accounts for 60% of the processing time across all the queries of the TPC-H benchmark.

Currently most relational databases are disk based where RAM is used as a temporary buffer for keeping recently or commonly referenced data. Typically a portion of the RAM space is reserved for join query processing, however this space may not beenough to fit any of the join relations in their entirety. In this case the join needs to be processed in parts, where each partprocesses a portion of the relations. Processing by parts usually means multiple passes through the data to complete the entire join.

The aim of efficient join processing in the above environment is to process the join by incurring the minimum number of read and write IO. Write IO is used to write out intermediate data to disk during join processing. There has been many different classic approaches for solving this problem, including, the nested loops join, sort-merge join, indexed-loops join, hash join, etc. The hash join is often found to be the most desirable among the alternative approaches, due to its ability to prune the number of comparisons without the need to first sort the data or build an index, both of which are expensive operations.

The classic GRACE and hybrid hash join methods [2] process the join in two phases. The first is the partitioning phase in which the input relations are scanned and written out into hash buckets. The second is the join phase where the two relations are joined one hash bucket at a time. The hybrid hash join differs from the above description by trying to join a subset of the tuples during the partitioning phase. It does this by keeping a memory resident hash bucket of the outer relation R during the partitioning phase. Then the tuples of the inner relation S that fall into the memory resident hash bucket are joined with the corresponding memory resident R tuples during the partitioning phase. This effectively filters out the tuples of the in-memory hash bucket from the join phase. It is better to make the outer relation R the smaller relation since this allows the in memory hash bucket occupy a larger percentage of total size of the outer relation.

Recently, Do et al. [3] performed a comprehensive performance evaluation of join algorithms on both hard disk drives and solid state drives. They found that the hybrid hash join was almost always the best performer on both types of secondary storage. However, the performance of the hybrid hash join can be significantly improved because the hybrid hash join does not make efficient use of the RAM space during the partitioning phase. The hybrid hash join does not consider the data characteristics of the two relations when it determines which tuples of R to keep in RAM during the partitioning phase. In contrast we take advantage of skewed join attribute distributions and missing values to magnify the benefit of joining tuples in the partitioning phase and thereby filter out a larger fraction of tuples from entering the join phase.

Skewed join attribute distributions occur in many real life situations. For example, Customers ⋈ C.CustomerID=O.CustomerID Orders. The number of orders per customer is usually skewed due to varied spending patterns of customers. For example, some customers buy regularly from a store whereas others shop rarely from the same store. Similarly the following join is likely to have skewed distributions, Orders ⋈O.ProductID=P.ProductID Products, because orders are typically of different sizes.

There can be many join attribute values from one relation which maybe missing from the other and vice versa. This can occur due to a selection being applied before the join for one or both of the relations. For example for σC.State=NY Customers ⋈C.CustomerID=O:CustomerID Orders, the selection on the Customers relation will likely mean that there are many orders in the Orders relation that will not find a match with the Customers relation.

In this paper we propose the filtered hash join (FHjoin) which filters tuples out during the partitioning phase of the hash join. FH-join uses two types of filters which are designed to take advantage of join attribute distribution skew and missing join attribute values. The first is a range filter which keeps in RAM R tuples that fall in certain join attribute ranges during the partitioning phase and then joining them directly with S's tuples as they are loaded during the partition phase. We keep the R range that simultaneously fit in RAM and also contains the largest number of S tuples. This allows us to take maximum advantage of skewed join attribute distributions. The second type of filter used is the bloom filter [4] which approximately records the presence or absence of individual join attribute values of the two relations. This is then used to filter out tuples of both R and S which have missing values in the opposing relation during the partitioning phase.

We performed an extensive performance study of FH-join against the hybrid hash join algorithm. The results show the FH-join outperforms hybrid hash join on both the total execution time and the total IO cost in all tested scenarios by up a factor of 4 for total execution time and by up to an order of magnitude for write IO when the data is highly skewed.

This paper makes the following key contributions: 1) introduced the key observation that exploiting join attributeskew and missing values can dramatically improve the performance of the hash join; 2) proposed the FH-join algorithm which modifies the hash join to exploit these data characteristics; and 3) performed an extensive experimental study comparing the FH-join against the hybrid hash join algorithm.

II. PRELIMINARES

In this paper we are interested in performing an equi-join between two relations R and S. Throughout the entire paper we will use R to denote the outer relation and S to denote the inner relation. |R| will be used denote the size of relation R in terms of the number of pages, and similarly for S. B is the size of a page. We will assume there is a RAM limit of M pages. The outer relation R will also be the smaller of the two input relations.

We will use the phrase **filtered out** extensively throughout this paper. We define it as follows: a tuple is filtered out if itis not written out into a disk resident hash bucket during the partitioning phase. A filtered out tuple is effectively joined during the partitioning phase of the hash join and therefore is pruned from the entering the join phase.

III.RELATED WORKS

There are four fundamental approaches for joining two relations. These include the block nested-loops join, sort merge join, Indexed-loops join and hash join. We will first briefly discuss these four approaches and then we will discuss advanced hash jointechniques. We give a more detailed discussion on the hybrid hash join [2] in Section 4, since it is the most similar join techniquecompared to our FH-join. See the paper by Mishra et. al.[5], Nooshin et.al. [6], and Balkesen et.al.[7] for a comprehensivesurvey of classical join techniques.

A. *Fundamental join System*

- **Block nested-loops join.** The block nested-loops join algorithm effectively processes the join using a nested for loop, where tuples are loaded into RAM at the block grain (multiple consecutive pages). The smaller relation is selected as the outer relation since these results in less passes through the inner relation. This approach performs comparisons between every pair of inner and outer relation tuples and therefore is very computationally expensive. If the outer relation is quite large compared to RAM size, then the block nested-loops join is very IO intensive since it needs to take one pass through the inner relation per chuck of the outer relation that fit in RAM.
- **Sort-merge join.** The sort-merge joins first sorts the two relations and then synchronously steps through the two relations in sorted order to perform the join. The sort-merge join performs less computation compared to the nested loops join since it uses the fact that the input relations are sorted to avoid comparisons between tuples that are far apart in sorted order. However, if the relations are not pre-sorted then the cost of performing the sort can be expensive [6].
- **Indexed-loops join.** In this method an index on the inner relation is used to process the join.

The join works by looping through the outer relation, while probing the inner relation using the index. The index on the inner relation allows a lot of comparisons to be pruned. Note random IO is incurred when probing the index; this is expensive for hard disk drives (HDDs) due to the high seek cost. This random IO cost can be significantly reduced with the effective buffering of pages of the inner relation. Theindexed-loops join is only really useful if there is a pre-built index on the inner relation. Building the index on the fly is veryexpensive due to the large number of random IOs.

- **Hash join.** The classic GRACE and hybrid hash join methods were described in the introduction. As already mentioned the strength of the hash join approach is that it can prune a lot of comparisons without the high pre-processing cost of sorting or building an index. Therefore it is often the preferred approach when the input relations are not sorted or pre-indexed [7].

B. *Advanced hash join techniques*

The focus of this paper is on improving the performance ofthe hash join algorithm. Therefore in this section we willreview some the more recent advances in the area of hashjoin algorithms.

Recent research [8, 9, 10, 11, 12, 13] has shownthat CPU processing occupies a significant portion of the total join processing time for the external hash join. This is due to the high cost of CPU cache misses and the fact the external hash join is very good at ensuring that all IO is sequential. Therefore a number of different approaches have been proposed to reduce CPU cache misses of the hash join. These techniques include pre-fetching [8], more temporal locality friendly reads and writes [10,11] and multi-threading [8, 9].

Boncz et al. [11] proposed the radix clustering approach for performing the hash partitioning in a CPU cache friendly way. The idea is to recursively partition data from course to finer and finer grained hash buckets. This approach reducesthe number of separate random locations written to at the same time. Therefore this approach is both more CPU cache and translation look aside buffer (TLB) friendly. The results show this approach can significantly reduce the partitioning cost of the hash join.

Chen et al. [10] proposed two techniques for speeding up the GRACE hash join. The first is called group pre-fetching and the second is called software-pipelined pre-fetching. The group pre-fetching technique apply modified forms of compiler transformations called strip mining and loop distribution to allow pipelining of consecutive probe tuples during the join phase. The results showed the proposed techniques can speed up the join phase by 2.0-2.9X and speedup the partitioning phase by 1.4-2.6X compared to the standard GRACE hash join.

Following the pre-fetching work, Chen et al. [12] developed the inspector join. Their idea is to gather statistics about the join attribute values of the two input relations during the partitioning phase in order to speed up the CPU performance of the join phase. They use the statistics in two ways: 1) create specialized filter indexes (multiple small bloom filters); and 2) decide which join phase algorithm to use for the specific data being joined. They want to avoid having to do the multiple repartitioning passes of the radix clustering algorithm by creating a separate bloom filter for each sub-partition of R. The sub-partitions are sized such that both a sub-partition and its created hash table fit in the CPU cache. Using the bloom filters they are able to tell which R sub-partition each S tuple belongs to. This idea is used to avoid writing to random locations when creating the hash table during the join phase. They also use the bloom filters to prune tuples of S that do not match any R tuples. Their work differs from ours in two respects. First, unlike us they do not take advantage of skewed data distributions to filter both R and S tuples during the partition phase. Second, they build their bloom filters during the partition phase in contrast we use pre-built bloom filters. The benefit of prebuilt bloom filters is that we can prune both R and S tuples during the partitioning phase. Whereas the approach of Chen et al. [12] which can only prune tuples of S during the partitioning phase.

Recently there has been a number of parallel hash join algorithms [8, 9] proposed. Kim et al. [8] compared the performance of sort merge join against hash join on modern multi-core CPUs. They found both join algorithms benefit greatly from multi-threading and sort-merge join benefits greatly from exploiting SIMD. They predict sort-merge join will outperform hash join when 512-bit SIMD instructions become available.

Blanas et al. [9] performed a thorough performance evaluation of existing hash join algorithms dissecting each internal phase and considering different alternatives for each phase. They found the partitioning phase does not benefit from multi-threading but the join phase can easily benefit from the added concurrency of multi-threading.

In contrast to all the above algorithms our algorithm is the only one to take advantage of data distribution and missing values to prune the number of tuples in both the R and S relations during the partitioning phase. Our approach can be used in conjunction with any of the above techniques to further improve hash join performance.

C. *Other earlier work on hash joins*

Gray et al. [14] proposed an adaptive hash join algorithm that is designed to work with dynamic changes in the available memory. Martin et al. [15] performed a detailed study comparing the hashed loops; GRACE and Hybrid hash join algorithms for multi-processor environments. Kitsuregawa et al. [16] performed a study into how best to the tune the size of the hash buckets for the GRACE hash join algorithm. Kitsuregawa et al. [17] proposed a load balancing strategy for parallel hash join algorithms. Kitsuregawa et al. [18] performed a study into the performance of the GRACE hash join algorithm for the parallel disk environment. DeWitt et al. [19] proposed skew handling methods for parallel joins.

None of the above techniques take data distribution or missing values into consideration when joining. The above techniques can all be enhanced by embedding our approach into their hash join algorithms.

IV. Analysis of Hybrid Hash Join

In this section we take a closer look at the hybrid hash join, in particular we will analyze the RAM buffer usage and IO costs of the hybrid hash join. We then identify how it can be improved.

Figure 1 shows how the limited RAM space is allocated to separate buffers during the partitioning phase of the hybrid hash join. One input buffer (I) is used to read in the input relations in large sequential chunks. One output buffer (O) is allocated to each of the disk resident hash buckets to enable sequential writing. Finally the remaining RAM space called the workspace (WS) is used to keep an in-memory hash bucket of R.

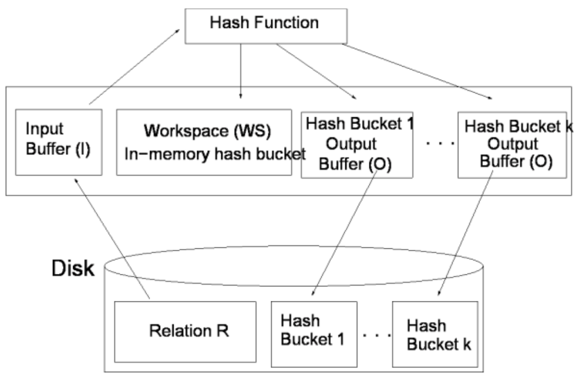

Hash Function
Input Buffer (I)
Workspace (WS) In−memory hash bucket
Hash Bucket 1 Output Buffer (O)
. . .
Hash Bucket k Output Buffer (O)
Disk
Relation R
Hash Bucket 1
. . .
Hash Bucket k

Fig.1 Partitioning the outer relation R in hybrid hash join

The partitioning phases of the hybrid hash join works as follows. First one chuck of the outer relation R is loaded into the input buffer. Next each tuple in the input tuple is hashed. If a tuple is hashed into the in-memory hash bucket then it is placed into a second in-memory hash table located in the workspace. Tuples that hash to other buckets are placed in its corresponding out buffer. Whenever an output buffer becomes full, its contents are flushed into the corresponding disk-resident hash bucket. This process is repeated until all the tuples in R have been partitioned.

The S relation is partitioned after R has been partitioned. The procedure for partitioning S is exactly the same as R except the tuples that hash into the in-memory hash bucket are immediately joined with the R tuple in the in-memory hash table. After S dataset has been partitioned. Then each of the disk-resident hash buckets of R and S are loaded in term into RAM and joined.

Keeping the in-memory hash bucket of R in the workspace allows all tuples that map into that bucket to be joined during the partitioning phase. There is a trade-off between assigning a larger versus smaller workspace. A larger workspace allows more tuples to be joined in the partitioning phase and thus reduces the number of IO spent writing during the partitioning phase and loading the disk resident hash buckets during the join phase. However a larger workspace also reduces the size of the input and output buffers, thus making IO less sequential.There has been research [20] into determining the optimal allocation of the RAM space to the various buffers.This approach effectively assumes that the data distribution of R and S is uniform and therefore cannottakeadvantage of possible skew in the distribution of S values. For example we can take advantage of the skew in the distribution of S values by keeping ranges in WS that correspond to a larger fraction of S tuples.

The hybrid hash join writes into a disk resident hash bucket any tuples that map into that bucket, this includes tuples that do not find a matching pair in the opposing relation. However, if we can somehow use a small part of WS to store information on the missing values of each relation, then we can use it to filter out the tuples during the partitioning phase that will definitely not find a match during the join phase.

V. Filtered Hash Join (FH-Join)

In this section we present our FH-join algorithm that improves over the hybrid hash join by making much more effective use of the workspace WS to filter out a larger fraction of R and S by taking advantage of skewed data distributions and missing join attribute values. The aim of FH-join is therefore to maximize the following objective:

$$\frac{|\text{TUPLES OF R } \mathit{filtered\ out}| + |\text{TUPLES OF S FILTERED OUT}|}{|WS|} \quad (1)$$

The above objective effectively says design an algorithm that filters out the most amounts of R and S per byte of WS available. We meet the above objective using two types of filters. The first is a range filter which selectively keeps attribute ranges in WS which filters out the largest fraction of S tuples. The second is a bloom filter that filters out tuples of R and S which do not have a matching pair in the opposing relation.

Figure 2 shows how the filters are used in the partitioning phase of FH-join to filter out tuples from the join phase.

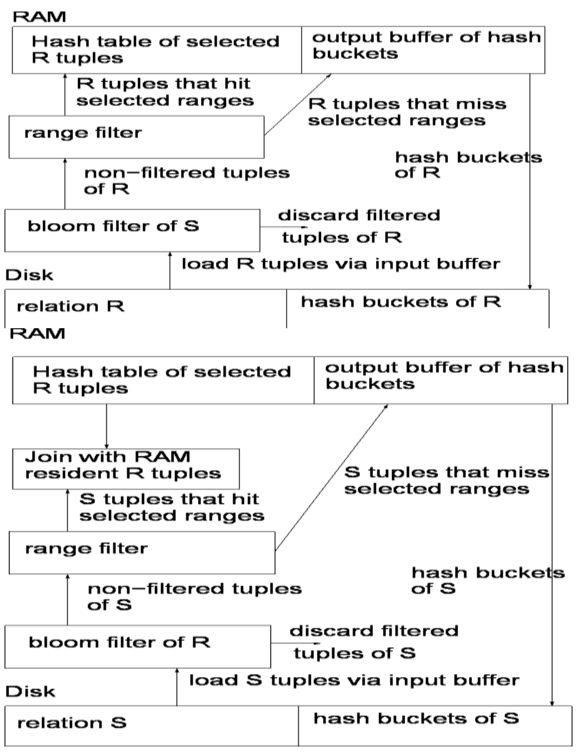

RAM
Hash table of selected R tuples
output buffer of hash buckets
R tuples that hit selected ranges
R tuples that miss selected ranges
range filter
hash buckets of R
non−filtered tuples of R
bloom filter of S
discard filtered tuples of R
load R tuples via input buffer
Disk
relation R
hash buckets of R
RAM
Hash table of selected R tuples
output buffer of hash buckets
Join with RAM resident R tuples
S tuples that miss selected ranges
S tuples that hit selected ranges
range filter
hash buckets of S
non−filtered tuples of S
bloom filter of R
discard filtered tuples of S
load S tuples via input buffer
Disk
relation S
hash buckets of S

Fig.2 Partitioning phase of FH-join for both R and S relations

Figure 5(a) shows how the tuples of R are filtered out. The tuples of R are loaded into the RAM via the input buffer. They are then compared against the bloom filter of relation S. This filtered out tuples of R with join attribute values that are determined to be missing by the bloom filter of S. The surviving non-filtered tuples are then stored in the in-memory hash table if they fall within the selected ranges of the range filter. Any tuples that pass through both of these filters are then placed into a disk resident hash bucket. Figure 5(b) shows the similar process for filteringS tuples. The difference is any tuples of S that fall within the selected ranges of the range filter are joined with the in-memory R tuples.

Algorithm 1 shows the high level algorithm of FH-join. The algorithm describes the pseudo code corresponding to the diagram in Figure 2. The algorithm uses histograms of R and S relations for creating the range filter. The histograms capture the frequency distribution of values within R and S. This is then used to select the value ranges of R which maps to a larger number of S tuples. The detailed algorithm for selecting ranges is described in Section 5.1. Algorithm 1 assumes the histograms and bloom filter of R and S have been pre-built. In Sections 5.4 and 5.6 we describe the cost of keeping the histograms and bloom filters up-to-date, respectively.

Algorithm1 HighlevelalgorithmofFH-join

```
1: constructrangefilterbasedonhistogramsofRandSrelations(Seesection5.1formoredetails)
2: usehistogramsofRandStoselectrangesforrangefilter
3: while more tuples in R need to be processed do
4:    Fillinput bufferwithtuplesofRloadedfromdisk
5: for each tuple rs of R inside the input buffer do
6:       if the bloomfilter of Sdetermines thatthe valueofrdoesnotexistinS then
7:          discardr
8:       elseif rangefilterjudgerisinselectedrangethen
9:          insertr intoanin-memoryhashtable
10: else
11:          writer intooutputbufferofhashbucket ofR
12:          if outputbufferbecomesfull then
13:             write outputbuffertocorrespondingdisk-basedhashbucket
14:          end if
15:       end if
16:    endfor
17: endwhile
18: while more tuples in S need to be processed do
19:    FillinputbufferwithtuplesofSloadedfromdisk
20:    for each tuplesofSinsidetheinputbufferdo
21:       ifthe bloom filterof Rdetermines that thevalue ofSdoesnotexistinRthen
22:          discards
23:       elseif ranges filterjudgesisinselectedrangethen
24:          join s with Rtuples in the in-memory hash table
25:       else
26:          writesintooutputbufferofhashbucketofs
27:          ifoutputbufferbecomesfullthen
28:             write outputbuffertocorrespondingdisk-basedhashbucket
29:          end if
30:       end if
31:    endfor
32: endwhile
```

FH-join only changes the partition phase of the hash join. Therefore the join phase is the same as GRACE and hybrid hash joins. Hence we focus our discussion in this section on the partitioning phase of the FH-join.

A. *Range filter*

The aim of the range filter is to make the best use of RAM by selecting the attribute ranges that cover the largest number of S tuples whilst making sure the R tuples that map into the selected ranges fit within a memory limit LRF. Given this aim it is best to select ranges where R is sparsely populated and S is densely populated. Figure 3 shows an example comparing two different attribute ranges, range A and rangeB. Each dot in the top row represents an R tuple and each dot in the bottom row represents an S tuple. In the example it is better to select rangeA than rangeB because rangeA filters out 12 S tuples whereas rangeB filters only 2 S tuples and both ranges consume the same amount of RAM space because they both contain the same number of R tuples. One reason for the success of rangeA is that it is sparsely populated in terms of R tuples, thereby allowing it to stretch over a longer interval of join attribute values. Therefore when selecting attribute ranges we need to consider the distribution of both R and S join attribute values.

We use histograms to capture the frequency distribution of join attribute values for both R and S. Any type of histogramcan be used. We will consider two popular histogram types: equi-width and equi-depth.

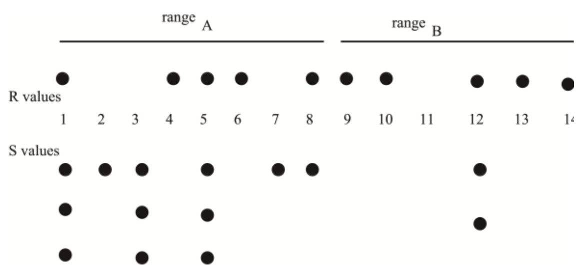


Fig.3 comparing two example ranges.

B. *Equi-width Histogram*

The length of the value range of an equi-width histogram is the same for all of its buckets. When using the equi-width histogram, we assign the same histogram bucket boundaries for all relations that may join with each other. We then select a subset of the histogram buckets that fits within LRF (R tuples of selected buckets fit within LRF), which also filters out the most number of S tuples.

No missing values and no duplicate values in R. If there is one tuple for each distinct value in R then finding the optimal set of histogram buckets is trivial since it just involves selecting the histogram buckets with the largest number of S tuples which also fits in LRF. This is because in this case each histogram bucket holds the same number of tuples. This trivial situation is quite common since the join attribute on R is often the primary key which often does not have anymissing or duplicate values.

R contains missing values and/or duplicate values. If there are missing and/or duplicate values in the joinattribute values of R then the problem becomes the NPcomplete knapsack problem. In the knapsack problem weare given a set of items I numbered 1 to n. Each item i $\in$ Ihas an integer size, si and has a value of vi. The aim is tofind the subset T $\subseteq$ I, such that $\sum_{i\in T}$ Si $\leq$ Land$\sum_{i\in T}$ Vi viis maximized, where L is a limit on the total size of itemsthat can fit within the knapsack. Mapping the knapsackproblem to our problem is trivial. The items map into thehistogram buckets, the size of each item maps to the numberof R tuples within the histogram bucket. This is becausethe R tuples are the ones that we are put into the RAM andthe RAM corresponds to the knapsack. The value of anitem maps into the number of S tuples within the histogrambucket. This is because we want to maximize the number ofS in the selected ranges (selected histogram buckets). Thesize limit L maps into our RAM limit LRF.

Having established that finding the optimal set of histograms is NP complete in the case of missing and/or duplicate join attribute values for R, we turn to the well-known greedy heuristic for solving the knapsack problem. In the greedy solution we first sort all the histogram buckets in terms of the profit (|SBi |/|RBi | ratio), where |SBi | and |RBi | are the number of tuples of S and R that map into bucket Bi, respectively. Then we try to fit as many buckets of R tuples as possible into RAM in descending order of profit. Algorithm 2 shows the pseudo code for the algorithm.

Algorithm 2 Greedy algorithms for selecting R join attribute ranges for the range filter when the equi-width histogram is used.

```
Input: ewHistR: equi-width histogram of relation R, ewHistS: equi-width histogram of relation S, LRF:
        RAM limit for storing R tuples of selected ranges
Output: selected Ranges: selected ranges of R
  1: Initialize selected Ranges to empty fjSBi j and
     jRBi j are the number of tuples of S and R that
     map into bucket Biof ewHistR and ewHistS,
     respectively
  2: Sort histogram buckets of ewHistR in terms of profit (jSBi j=jRBi j ratio) and place in sorted bucket Array
  3: for each bucket b within sorted bucket Array in descending order profit do
  4: if (total size of tuples within selected Ranges + total size of R tuples in b) < LRF then
  5: Place value range of b into selected Ranges
  6: end if
  7:end for
```

C. *Equi-depth Histogram*

The equi-depth histogram has variable bucket boundaries but fixed histogram height (the number of items that map into the same bucket). Therefore each bucket contains the same number of items. One of the major benefits of the equi-depth histogram is that it is much more sensitive to the value distribution. Therefore there will be more buckets in value ranges that have higher number of tuples.

Like for the equi-width histogram case here we also aim to select the subset of histogram buckets that fits within LRF, and filters out the most number of S tuples. We again first consider the case of no missing or duplicate R join attribute values.

No missing values and no duplicate values in R. In the case of no missing and/or duplicate R join attribute values the solution is trivial. We just select the set of S histogram buckets which have the smallest width and which also fit within LRF. This is because each bucket has the same number of S tuples but the smaller width buckets covers a smaller number of R tuples. Therefore if we select the smaller width buckets we can fit more buckets within LRF.

R contains missing values not due to prior selection and/or contains duplication. This case is less trivial, since the equi-depth histogram buckets of R and S would not be aligned and therefore we need to interpolate the number of tuples in one of the histogram buckets in order to effectively align the buckets. We cannot select a subset of S histogram buckets like the previous two cases because we would then need to interpolate the R histogram buckets to approximate the number of R tuples that would fall into the selected S tuples. This approximation would mean we cannot guarantee to stay within the memory limit of LRF. Hence we select a subset of the R histogram buckets such that selected buckets stay within the memory limit and the interpolated number of tuples in the corresponding S histogram buckets is the largest. The reason this is not an NP complete problem like the corresponding equi-width histogram case is that all the R buckets have the same number of tuples. Therefore it is not the Knapsack problems because it effectively means each item has the same weight.

For a given join attribute value range v we can compute the number of tuples in the corresponding range of S by interpolating the values of the S histogram buckets, NTS (v) as follows:

$$NTs(v) = \sum_{b \in OVERLAPBs\,(v)} \left(\frac{OVERLAP\ Lengt\ h(b,v)}{lengt\ h(b)} \right) NT(b) (2)$$

Where OVERLAPBS (v) is the set of S histogrambuckets that overlap v, OVERLAP Length (b; v) is thelength of the overlap between the value range of bucket band v, length (b) is the length of value range of bucket b,and NT(b) is the number of tuples in bucket b.Using Equation 2 we can compute the number of tuples ofS that fall within the value range of a R histogram bucket.

D. Equi-width versus equi-depth histograms

In this section we compare the equi-width and equi-depth histograms in terms of update cost, quality of selected ranges and run-time complexity.Where quality of a selected range is measured in terms of a higher fraction of R and S tuples filtered out per byte of workspace WS (Equation 1).

Update costs. The main benefit of using an equi-width histogram is that updating the histogram is trivial and cheap. This is because its bucket boundaries do not change with respect to updates. Inserting a tuple simply requires incrementing the corresponding bucket count and deleting a tuple simply requires decrementing the corresponding bucket count. However, incrementally updating an equi-depth histogram is very difficult since updates result in changes to multiple bucket boundaries. This may not be a big problem for applications that are mostly read-only such as data warehouses.

Quality of selected ranges. For the case that R contains no missing values and no duplicated values the equi-depth histogram will provide higher quality selected ranges because it is much more sensitive to skew in the number of tuples in both R and S. As mentioned before the equi-depth histogram would have more histogram buckets in ranges that have higher number of tuples. For the case that R contain no duplicate values but contain missing values due to prior selection, again the equi-depth histogram will produce higher quality selected ranges. This is for the same reason as the first case. However, for the last case where R contains missing values not due to prior selection and/or contains duplication, it is less clear the equi-depth histogram is better than the equi-width histogram. The reason for this is the selection of value ranges is based on the bucket boundaries of R instead of bucket ranges of S. Therefore we cannot take advantage of finer grained ranges in regions that have more S values. Selecting based on the attribute boundaries of S is more important than R since S is the inner relation and therefore should be larger in size. The other thing to consider for this third case is that both equi-width and equi-depth histogram solutions are approximate solutions. For the equi-width histogram it is a greedy heuristic solution to the knapsack problem and forthe equi-depth histogram interpolation is used to approximate the number of tuples S tu (2) ın each range of R.

Run-timecomplexity. All the proposed algorithms for both types of histograms have run-time complexity of O(nlog(n)), where n is the number of histogram buckets in either R or S depending on the algorithm. This is because they all involve first sorting the ranges corresponding to R or S histogram buckets and then selecting the subset of ranges based on sorted order. Although when the equi-width histogram is used for the case that R contain missing values and/or duplicate values is an NP complete problem, we proposed the use of the greedy algorithm which again has a run-time complexity of O(nlog(n)).

Recommendation. Based on the aboveanalysiswemakethefollowingrecommendations .Use the equi-width histogram whenever the data is expected to be updated fairly frequently due to the high cost of updating equi-depth histograms. However, if the data is mostly read-only then the equi-depth histogram should be used except the third case where R contains missing values not due to prior selection and/or contains duplication. In this third case there is no clear winner between equi-width and equi-depth histograms. These recommendations are summarized in Table 1.

TABLE I

Recommendations on when the equi-width versus equi-depth histogram should be used

Missingand/orduplicate valuesin*R*	**Mostlyread -only**	**Equi-width**	**Equi-depth**
Nomissingandnoduplicate	Yes		✓
Nomissingandnoduplicate	No	✓	
Onlymissingduetopriorselection	Yes		✓
Onlymissingduetopriorselection	No	✓	
Missingnotduetopriorselectionand/o	Yes	✓	✓
Missingnotduetopriorselectionand/o	No	✓	

E. *Bloom Filter*

If we know which join attribute values do not exist for the opposing relation then we can directly discard any tuples that map to a join attribute value that does not exist in the opposing relation. One naive way to store which join attribute values exist within a relation is to use 1 bit per join attribute value. However, this approach would consume a lot of RAM space since the join attribute values may span a large domain. In this paper we use a bloom filter [4] as effectively a lossy compression method to store and index which join attribute values that exist within a relation. The bloom filter can be set to any size and has the desirable property that it will never produce any false negatives. That is if a join attribute value is found to not exist in the bloom filter then it is guaranteed to not exist in the relation. However, a false positive is possible, that is, a join attribute value that exists within the bloom filter may not actually exist in the relation. False positives do not result in missing join results but just means that the false positive tuple cannot be filtered out by the bloom filter.

The bloom filter works by first hashing the join attribute values of one relation and then using the hash value as an index into a bit array. Next, the cell within the bit array that corresponds to the hash code is set to 1 indicating the existence of at least one tuple whose join attribute value hashes to that location. The filter can then be probed by hashing join attribute values of the opposing relation and then using the hash value to index into the corresponding cell of the bit array. If the indexed cell is zero then it means there is no tuple with that join attribute value in the opposing relation. Hence the tuple can be safely filtered out.

Figure 4 shows an example of using the bloom filter to store the existence of S tuples. In the example the set of S join attribute values 20, 4, 22, 1, 18 are inserted into the bit array of the bloom filter by using their corresponding hash values 1, 2, 4, 9 and 4. The bit array positions 1, 2, 4 and 9 are set to 1 accordingly. Next the R join attribute value 5 is used to probe the bloom filter by computing the hash value of 5, which is 8 in this case. The eighth position in the bit array is inspected. A bit value of 0 is found at that location, indicating the join attribute value of 5 does not exist in R. Therefore the tuple of R with join attribute value of 5 can be safely filtered out. In contrast the join attribute value of 66 when used to probe the bit array results in a false positive since the hash value of 66 is 9 which maps into a location of the bit array which has 1 a bit set. This is a false positive since the set of S values does not contain 66. The 9th element of the bit array was set to 1 since the value 1 also has a hash value of 9.

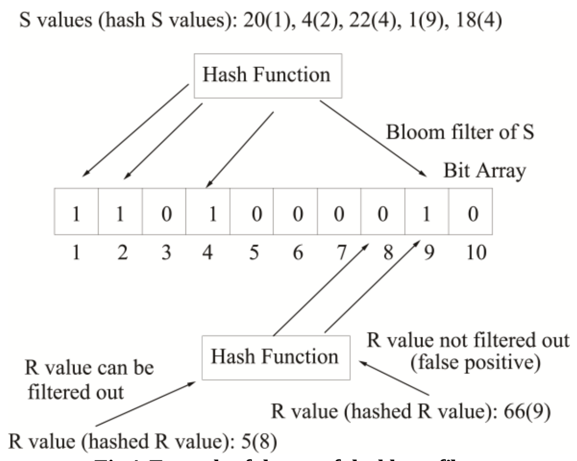

S values (hash S values): 20(1), 4(2), 22(4), 1(9), 18(4)
Hash Function
Bloom filter of S
Bit Array
1 1 0 1 0 0 0 0 1 0
1 2 3 4 5 6 7 8 9 10
Hash Function
R value not filtered out (false positive)
R value can be filtered out
R value (hashed R value): 66(9)
R value (hashed R value): 5(8)

Fig 4. Example of the use of the bloom filter

Reducing false positives. As mentioned earlier false positives does not produce incorrect join results since it just means the bloom filter cannot filter out the false positive tuples. However, false positives are undesirable since they reduce the number of tuples of S that can be filtered out during the partitioning phase. There are two approaches for reducing the frequency of false positives. First, the bit array can be enlarged. This results in less hash collisions. The second approach is to use multiple hash functions. In the case multiple hash functions are used, a false positive only occurs when a value results in a hash collision for all hash functions used. The first approach of enlarging the bit array reduces false positives by using more RAM space, whereas the second approach of using multiple hash functions reduces false positives by using more computation.

Selection prior to join. In the situation there is a selection prior to a join; a bloom filter can be built on-the-fly for the selected tuples during the selection operation.

F. Keeping bloom filters up-to-date

When tuples are deleted we cannot simply reset its corresponding entry in the bit array to 0 since multiple values can hash to the same location in the bit array. However, doing nothing when a tuple is deleted does not produce an incorrect join result since it just generates a false positive. Therefore we do not update the bloom filter when tuples are deleted, but instead rebuild the entire bloom filter after a threshold number of tuples have been deleted. Insertion into the bloom filter is cheap and straight forward as described throughout this section. It is also important to note that there are a lot of applications which are mostly append only like most datawarehouses. For these applications keeping the bloom filter up-to-date will not be a problem since no deletions are needed.

VI. Experimental setup

The experiments were conducted using a 500GB Seagate ST3500418AS 3.5 inch SATA Hard Drive. The processor we employed in the experiments is an Intel(R) Core(TM) i7 CPU 860. The machine had 4 GB of RAM. However, we further restricted the amount of RAM available to the tested algorithms. The experiments were conducted on the Linux operating system. Linux automatically caches all IO requests. This would invalidate our experiment results since it would mean pages loaded during the partitioning phase will be available for reuse in the join phase without the need to reload from the hard disk. Therefore, we disabled the operating system's caching functionality.

A. Algorithms tested

HH-Join. This is the traditional hybrid hash join. We set the size of the input buffer, output buffer, working space and the number of disk resident hash buckets using the formulas developed by Hass et al. [20] (as explained in Section 4). The universal fudge factor was set to the recommended 1.2.

FH-join. For the FH-join, we set the size of the input and all the output buffers to 700 pages (2.8 MB) each and the number of disk resident hash buckets to 50. These parameters were determined based on experimental tuning. We leave the work of finding optimal allocation of buffer sizes for FH-join as an area of future work. We set the size of the bloom filter to 164 KB. We set the memory limit for the range filterLRF to equal the size of the work space minus the size of bloom filter. In the experiments we used an equal-depth histogram.

B. Data sets

In our experiments the size of the tuples of R and S relations were modelled using the ORDERS and LINEITEMS tables of the TPC-H benchmark. We also modelled the join attribute as a 32-bit integer representing the ORDERKEY. We varied the ratio of R relation size versus S relation size instead of just using the one defined by TPC-H. This is because our algorithms are highly sensitive to this ratio and it is important to test a range of ratios. We also did not use the exact data distribution specified by the TPC-H benchmark since we wanted to test a range of join attribute value distributions.

We generated three data sets. The three data sets are described as follows. Complete-uniform, where the join attribute of R have no missing values and no duplicates. The join attribute values of S are generated using the uniform random distribution. Complete-Gaussian, where the join attribute of R have no missing values and no duplicates. The join attribute values of S are generated using the Gaussian random distribution. We use complete-Gaussian as the default data distribution because this is a common scenario. In this common scenario the join attribute of R is a primary key and the join attribute of S is a foreign key. The primary key cannot have any duplicate values and often has the complete set of values. The join attribute of the foreign key can often be skewed. Gaussian-Gaussian, where both the join attributes of both R and S datasets are generated using the Gaussian random distribution. Table 2 shows the default settings used in our experiments.

		Defaultsetting
RAMsize(pages)		80000
Percentageof$R$valuesmissing		0
σofGaussiandistribution		0.5
Sizeof$S$relation(pages)		2857369
Sizeof$R$relation(pages)		245964
Tuplesizeof$S$(bytes)		112
Tuplesizeof$R$(bytes)		104
Datadistribution		Complete-Gaussian

TABLE III

Default parameter setting used in our experiments

VII. Experimental results

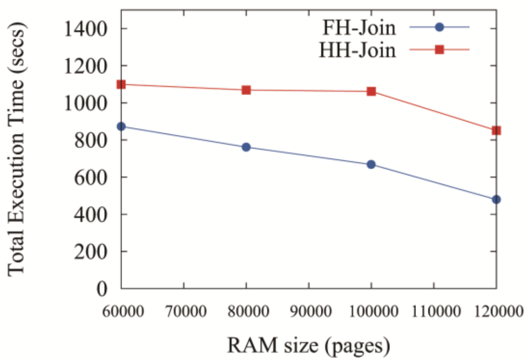
We have conducted six experiments. In the first experiment, we have varied the RAM size. In the second experiment, we have varied the percentage of values missing from R. In the third experiment, we have varied the level of skew in S. In the fourth experiment, we varied the size of S. In the fifth experiment, we reported the breakdown of execution time for three different RAM sizes. Finally, we varied the data distribution of both R and S.

A. *Vary RAM size results*

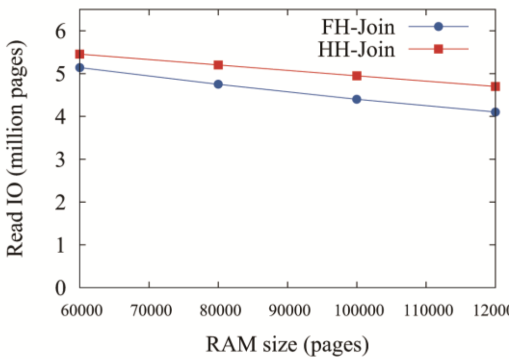
In this experiment, we compare the performance of the FH-join and HH-join with varying RAM sizes and the rest of the parameters set using the default parameters.

Figure 5(a) shows the total execution time result when the RAM size is varied. Figure 5(b) and Figure 5(c) shows the read and write IO results, respectively. Figure 5(d) shows the percentage of S tuples filtered out during the partition phase of the two competing hash join algorithms.

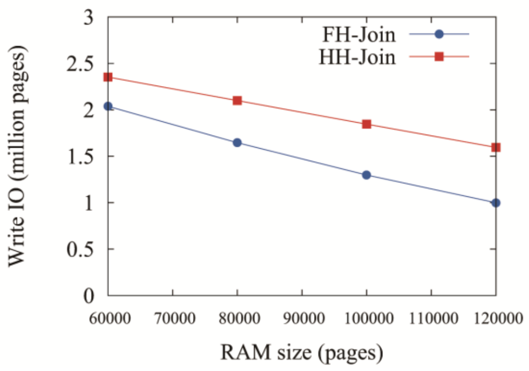
Figure 5(a) shows that the FH-join algorithm significantly outperforms the HH-join algorithm for total execution time and the percentage difference between the algorithms increase as the RAM size increases. This is because the FH-join algorithm makes more effective use of RAM by analysing the skewed distribution of S and then retaining the ranges of R which filters out the highest percentage of S tuples. This can be seen from Figure 5(d). The main effect of filtering out more tuples is the dramatically smaller number of write IOs of the FH-join compared to the HH-join as shown on Figure 5(c).



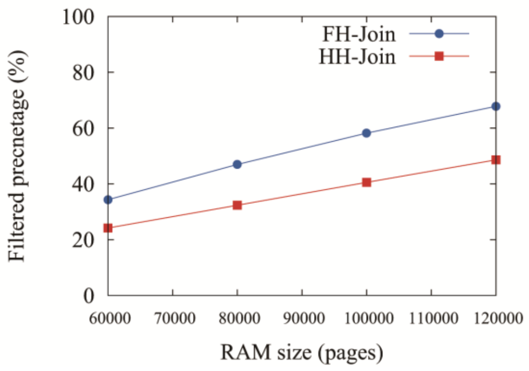
(a) Total execution time



(b) Number of read IO



(c) Number of Write IO



(d) Filtered Percentage

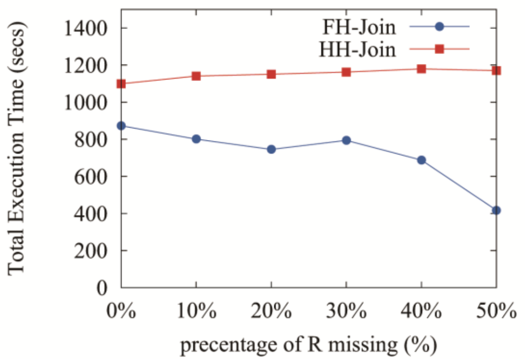
Fig 5. Results of the varying RAM size experiment.

B. *Vary percentage of missing R values*

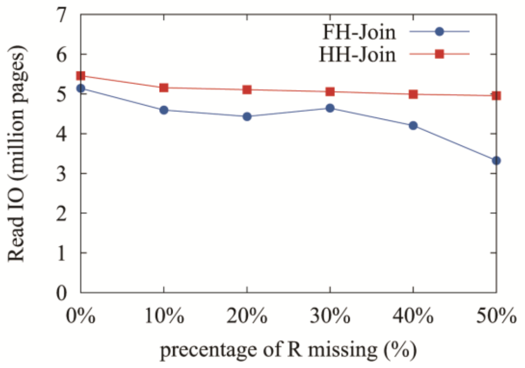
In this experiment we compare the performance of FH-join and HH-join when the percentage of R tuples missing is varied. We randomly (using uniform random distribution) remove between 0% and 50% of the values of relation R. We left the other parameters the same as the default settings.

Figure 6(a) shows the total execution time performance of the FH-join compared to the HH-join. The results indicate that the FH-join outperforms HH-join for the whole range of percentage of missing R tuples varied. In particular FH-join outperforms HH-Join by a factor of 3 for total execution time and a factor of 5 for write IO when 50% of the R tuples are missing.

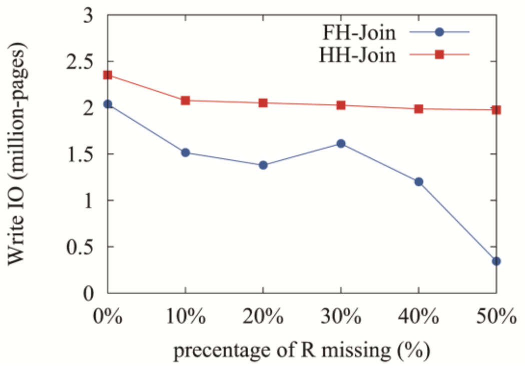
It can be clearly seen from the graph that the FH-join outperforms the HH-join by a larger margin as the percentage of R tuples missing increases. This is because the FH-join uses a bloom filter to filter out tuples of S which map to the missing R values, whereas the HH-join does not keep track of which R values are missing and therefore cannotfilter out the corresponding S tuples. Figure 6(d) clearly shows the effectiveness of the bloom filter at filtering out S tuples, when the percentage of R tuples removed is 50% the two filters of the FH-join combine to remove 80% of the tuples of S. In contrast the percentage of tuples filtered out by the hybrid hash join stays constant since it does not know which R tuples are missing.



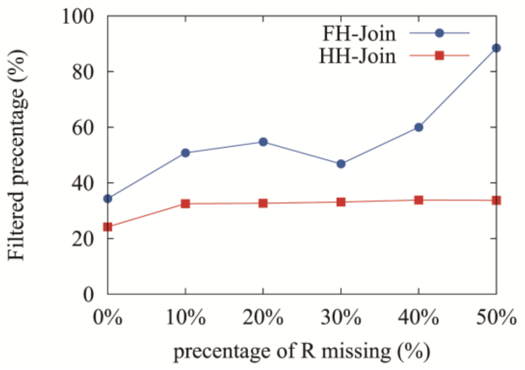
(a) Total execution time



(b) Number of read IO



(c) Number of Write IO



(d) Filtered Percentage

Fig 6. Results of percentage of missing R tuples

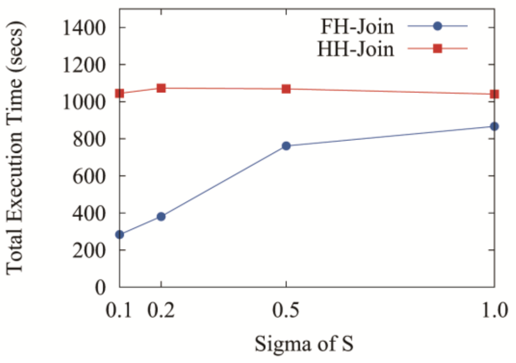
C. Vary degree of skew in S

In this section, we compare the performance of the two algorithms when the skew in the values of S is varied by varying the sigma value of the Gaussian distribution from 0.1 to 1.0. A larger sigma value means smaller skew. Therefore the graphs show the results from higher skew to lower skew.

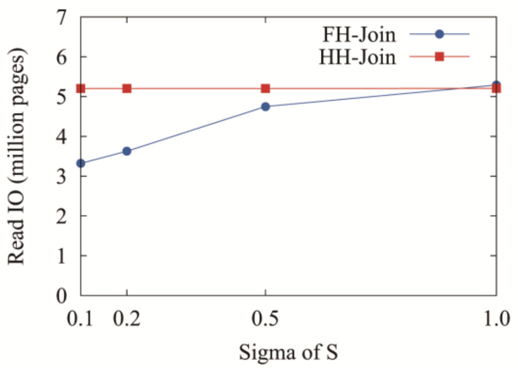
Figure 7 shows, that the FH-join outperforms HH-join algorithm on both the total execution time and the total IO cost in all tested scenarios by up a factor of 4 for total execution time and by up to an order of magnitude for write IO when the data is highly skewed (sigma equal to 0.1). This is because the range filter of FH-join is more effective (prunes a larger percentage of S tuples) when the data distribution of S is more skewed.

The results show FH-join gets closer to the performance of the HH-join as the degree of skew reduces. This is because as the degree of skew decreases FH-join loses more of its advantage of exploiting skew to filter out more S tuples. Hence it performs more similar to HH-join as the degree of skew decreases. This is supported by the results of Figure 7(d) which shows the percentage of tuples filtered out by the FH-join is about the same as HH-join when sigma is at 1.0 (lowest skew).

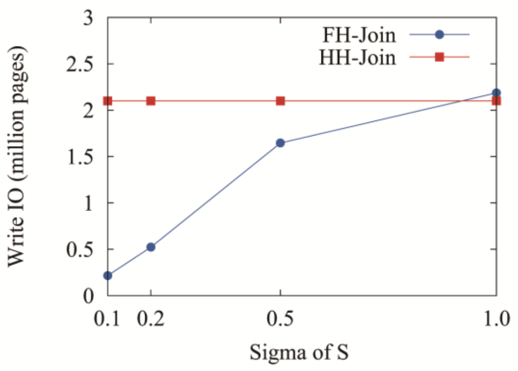
The results show that although FH-join filters out around the same percentage of tuples at sigma of 1.0 (shown in Figure 7(d)) as the HH-join FH-join however outperforms HH-join for total time by a noticeable margin (Figure 7(a)). This is because our FH-join implementation makes very efficient use of the CPU when matching tuples in the partitioning phase by using knowledge that the R relation does not have any missing values and it is ordered in RAM.



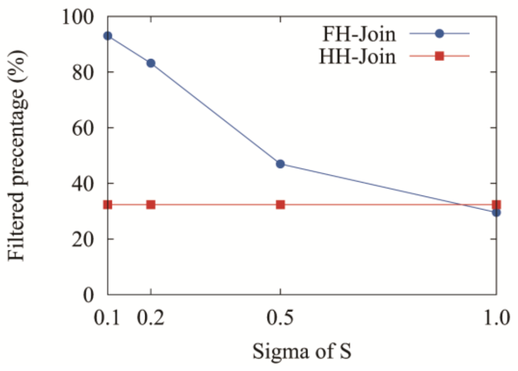
(a) Total execution time



(b) Number of read IO



(c) Number of write IO



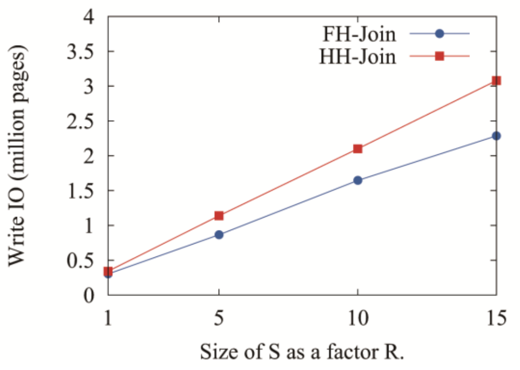
(d) Filtered Percentage

Fig 7. Results of varying skew in S.

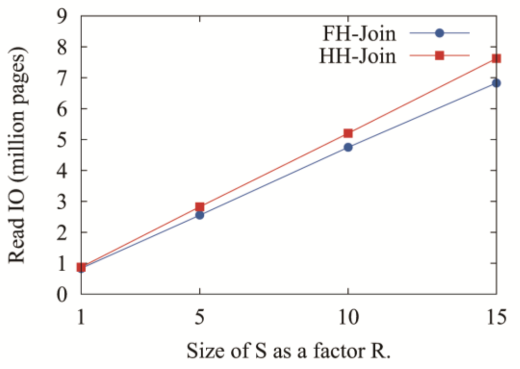
D. *Vary size of S*

In this experiment, we compare the performance of the FH-join and the HH-join with varying sizes of S. The size of relation S is labelled in terms of the number of factors by which the S relation is larger than the R relation. We left other parameters to default settings.

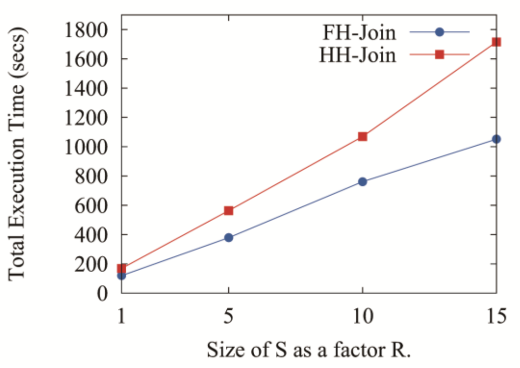
The results in Figure 8 show that the FH-join outperforms HH-join by a larger amount as the size of relation S grows. The reason is again due to the fact that the FH-join uses RAM more efficiently during the partitioning phase. Each good decision of which ranges of relation R to keep in RAM is magnified as the size of S grows, hence increasing the gap between the two algorithms as the size of S grows.



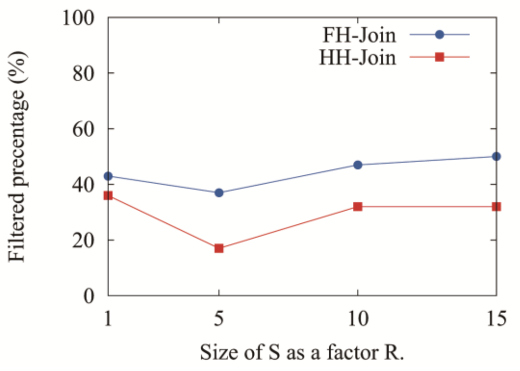
(a) Total execution time



(b) Number of read IO



(c) Number of write IO



(d) Filtered Percentage

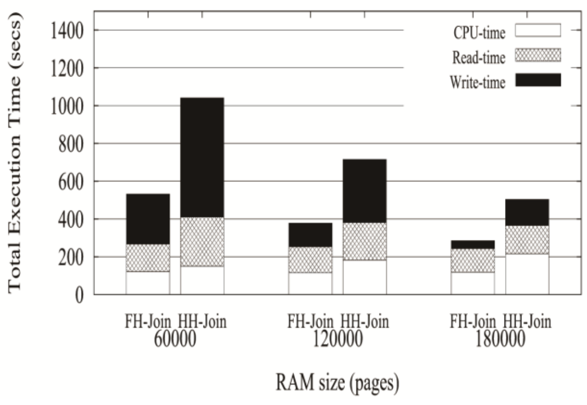
Fig 8. Results of varying the size of S.

E. Breakdown of total execution time as RAM size is varied.

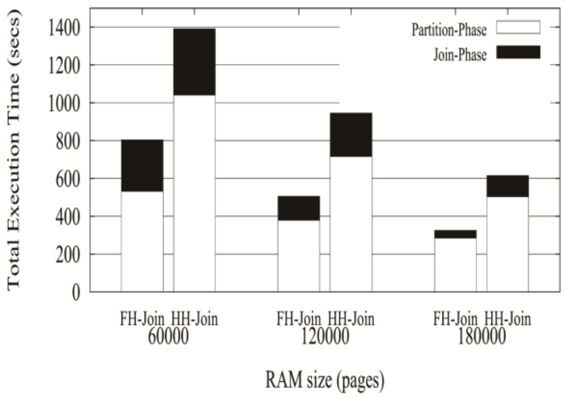
In this section we measure the execution time of the two join algorithms by breaking down the total execution time into two categories. Firstly, the execution time is divided into two parts, the partitioning phase, and the join phase. Secondly, the execution time is divided into three parts, the write IO time, the read IO time, and the CPU time. For each breakdown graph we report the results for three different RAM sizes.

The partitioning and join phase breakdown results are reported in Figure 9(a). As can be seen from the graph the partitioning phase consumes a much higher percentage of the total time compared to the join phase. This is because both algorithms are able to complete the join of a high percentage of tuples during the partition phase and therefore leaving few tuples left to join during the join phase. Also the non-filtered out tuples are written out during the partition phase which contributes to a significant amount of execution time during the partitioning phase (see Figure 9(b)).

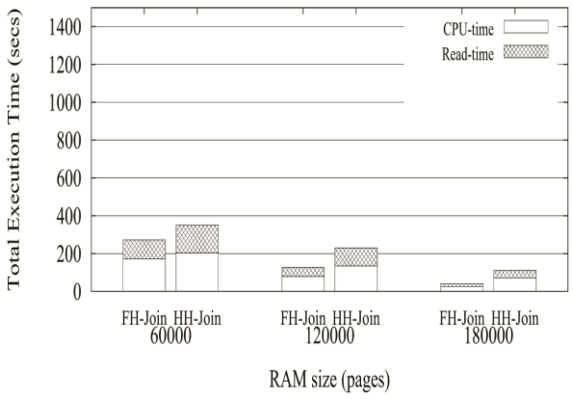
The results of the breakdown between read IO time, write IO time and CPU time for the partitioning phase is shown in Figure 9(b). The results show as the RAM size grows both algorithms spend less time performing write IO. This is because as the RAM size grows a larger amount of relation R can be fit in memory for both algorithms and can therefore be used to filter out a larger amount of S tuples which in turn reduces the need to write out tuples for the join phase. The results of the breakdown between read IO time and CPU time for the join phase is shown in Figure 9(c). We do not report write IO times for the join phase since the join phase does not need to perform any write IO. The results show CPU time is a larger portion of total execution time than read IO time for the join phase. The reason for this is the join phase is CPU intensive due to the large number of comparisons it needs to perform and also the large number of random RAM accesses during hash table creation and probing.



(a) Total execution time



(b) Partition phase



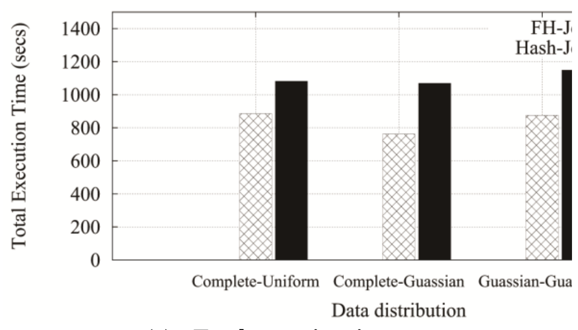
(c) Join phase

Fig 9. Results of breakdown of total execution time as RAM size is varied

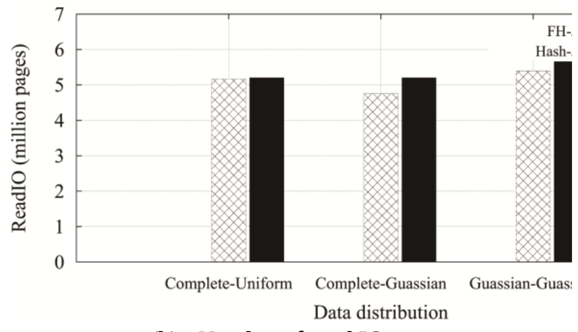
F. Vary data distributions of R and S

So far, in previous experiments, we used the Complete-Gaussian data distribution data set. In this section, we report results for data distributions of Complete-Uniform, Complete-Gaussian and Gaussian-Gaussian (see Figure 6.2 for a description). The other parameters were left at their default values.

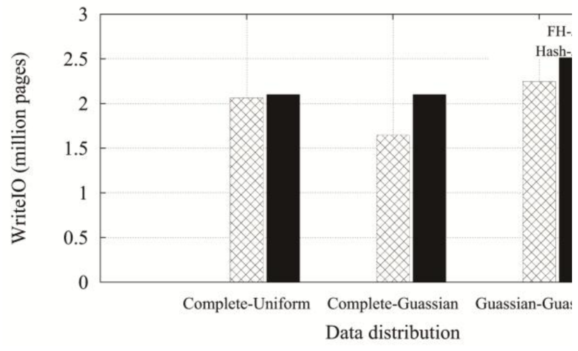
Figure 10 shows the results for this experiment. As shown in the results, the FH-join outperforms the HH-join for all of the three vary distribution dataset groups for total execution time. Among these three tested data distribution groups, FH-join gives the best performance in Complete-Gaussian distribution. This is because FH-join can take advantage of the skew in the S relation to filter out more tuples during the partitioning phase. In addition it also uses the knowledge that R has no missing tuples and is ordered in terms of join attribute values to join tuples faster during the join phase.



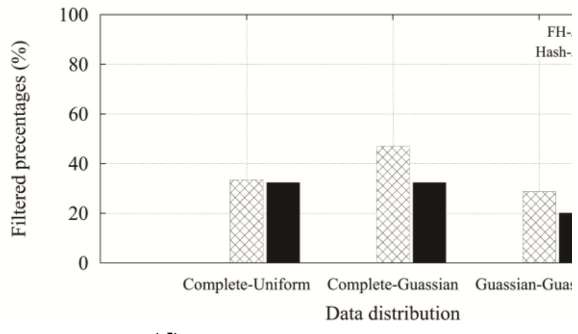
(a) Total execution time



(b) Number of read IO



(c) Number of write IO



(d) Filtered percentage

Fig 10. Results of varying data distribution of R and S

VIII. CONCLUSION

In this paper, we proposed a new approach for speeding up the partitioning phase of the external hash join algorithm, by reducing the total read and write IO costs. The approach is designed to make the best use of the limited RAM space available during the partitioning phase to filter out as many tuples both relations as possible from entering the join phase. The approach is called FH-join. The FH-join uses a range filter and a bloom filter to prune the number of tuples entering the join phase.

A detailed experimental study was conducted into the effectiveness of the FH-join against the hybrid hash join algorithm. The results show that the FH-join algorithm outperforms the hybrid hash join algorithm in terms of both total execution time and total IO cost in almost all scenarios tested. The FH-join algorithm outperforms the hybrid hash join by a larger margin as any one of the following happens: RAM size increases; degree of skew in the attribute value of the outer relation increases; the percentage ofmissing values increases;and as the size of the inner relation increases. The results also showed that the FH-join gets most of its performance advantage against hybrid hash join from reducing write IO during the partitioning phase of the join.

In the future, we plan to propose more effective methods for selecting the best ranges during the partitioning phase and further optimize the CPU performance of the FH-join algorithm. We also plan to explore the performance implications of using the FH-join in a multi-threaded environment.

REFERENCES

[1] Hayes, T., Palomar, O., Unsal, O., Cristal, A., and Valero, M. (2012) Vector Extensions for Decision Support DBMS Acceleration. The 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO45), pp. 166–176.

[2] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. A. (1984) Implementation techniques for main memory database systems. Proceedings ACM SIGMOD '84, pp. 1–8.

[3] Do, J. and Patel, J. M. (2009) Join processing for flash ssds: remembering past lessons. DaMoN, pp. 1–8.

[4] Mullin, J. K. (1983) A second look at bloom filters. Communications of the ACM, 26, 570–571.

[5] Mishra, P. and Eich, M. H. (1992). Join processing in relational databases. ACM Computing Survey, 24, 63–113.

[6] Nooshin S, Mirzadeh, Kockerber, O, Falsafi B, and Grot B. (2015) Sort vs. Hash join revisited for near-memory execution, pp. 1-6

[7] Balkesen C, Teubner J, Alonso G, et al. (2015). Main-Memory Hash Joins on Modern Processor Architectures [J]. IEEE Transactions on Knowledge and Data Engineering, , 27(7): 1754-1766.

[8] Kim, C., Kaldewey, T., Lee, V. W., Sedlar, E., Nguyen, A. D., Satish, N., Chhugani, J., Di Blas, A., and Dubey, P. (2009) Sort vs. hash revisited: fast join implementation on modern multi-core cpus. Proceedings of the VLDB, 2, 1378–1389.

[9] Blanas, S., Li, Y., and Patel, J. M. (2011) Design and evaluation of main memory hash join algorithms for multicore cpus. SIGMOD Conference, pp. 37–48.

[10] Chen, S., Ailamaki, A., Gibbons, P. B., and Mowry, T. C (2004) Improving hash join performance through prefetching. Proceedings of the 20th International Conference on Data Engineering ICDE '04, pp. 116–127.

[11] Manegold, S., Boncz, P. A., and Kersten, M. L. (2000) What happens during a join? Dissecting CPU and memory optimization effects. Proceedings of the VLDB, pp. 339–350.

[12] Chen, S., Ailamaki, A., Gibbons, P. B., and Mowry, T. C. (2005) Inspector joins. Proceedings of VLDB, pp. 817–828.

[13] Manegold, S., Boncz, P., and Kersten, M. (2002) Optimizing main-memory join on modern hardware. IEEE Transactions on Knowledge and Data Engineering, 14, 709–730.

[14] Zeller, H. and Gray, J. (1990) An adaptive hash join algorithm for multiuser environments. In McLeod, D., Sacks-Davis, R., and Schek, H.-J. (eds.), Proceedings of VLDB, pp. 186–197.

[15] Martin, P., Larson, P.-A°., and Deshpande, V. (1994) Paralle hash-based join algorithms for a shared-everything. IEEE Trans. Knowl. Data Eng., 6, 750–763.

[16] Kitsuregawa, M., Nakayama, M., and Takagi, M. (1989) The effect of bucket size tuning in the dynamic hybrid grace hashjoin method. VLDB, pp. 257–266.

[17] Kitsuregawa M, Ogawa Y. Bucket spreading parallel hash: a new, robust, parallel hash join method for data skew in the super database computer (SDC)[J]. Very large data bases, 1990: 210-221.

[18] Kitsuregawa, M., ichiro Tsudaka, S., and Nakano, M. (1992) Parallel grace hash join on shared-everything multiprocessor:Implementation and performance evaluation on symmetry s81. Proceedings of ICDE.

[19] DeWitt, D. J., Naughton, J. F., Schneider, D. A., and Seshadri, S. (1992) Practical skew handling in parallel joins. Proceedings of VLDB, pp. 27–40.

[20] Haas, L. M., Carey, M. J., Livny, M., and Shukla, A. (1997) Seeking the truth about ad hoc join costs. The VLDB Journal, 6, 241–256