MCJoin: A Memory-Constrained Join for Column-Store Main-Memory Databases.

Steven Begley
Department of Computer Science and Computer Engineering
La Trobe University
Melbourne, Victoria, Australia
s.begley@latrobe.edu.au

Zhen He
Department of Computer Science and Computer Engineering
La Trobe University
Melbourne, Victoria, Australia
z.he@latrobe.edu.au

Yi-Ping Phoebe Chen
Department of Computer Science and Computer Engineering
La Trobe University
Melbourne, Victoria, Australia
phoebe.chen@latrobe.edu.au

ABSTRACT

There exists a need for high performance, read-only main-memory database systems for OLAP-style application scenarios. Most of the existing works in this area are centered around the domain of column-store databases, which are particularly well suited to OLAP-style scenarios and have been shown to overcome the memory bottleneck issues that have been found to hinder the more traditional row-store database systems. One of the main database operations these systems are focused on optimizing is the JOIN operation. However, all these existing systems use join algorithms that are designed with the unrealistic assumption that there is unlimited temporary memory available to perform the join. In contrast, we propose a Memory Constrained Join algorithm (MCJoin) which is both high performing and also performs all of its operations within a tight given memory constraint. Extensive experimental results show that MCJoin outperforms a naive memory constrained version of the state-of-the-art Radix-Clustered Hash Join algorithm in all of the situations tested, with margins of up to almost 500%.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing, Relational Database*

General Terms

Algorithms, Performance

Keywords

Hash Join, Main Memory, Column Store

1. INTRODUCTION

There exists particular application domains for which there is a growing need for high-performance, read-only database systems. Scenarios include online analytical processing (OLAP) and scientific databases, which strongly favor database read operations over write operations. OLAP encompasses fields such as marketing, budgeting, management and financial reporting, and other business intelligence activities where the ability to rapidly explore vast volumes of historical data while looking for trends and patterns can be of critical advantage. So too, scientific databases may need huge amounts of collected data to be analyzed in an expeditious manner.

The continued increases in RAM density, coupled with reductions of RAM prices, have resulted in main-memory databases becoming more common for OLAP scenarios. Consequently, there has been much research towards developing main-memory databases, resulting in such systems as MonetDB [6] and C-Store [18]. These systems have been tailored for such main memory operations by employing novel techniques (such as storing tables column-wise instead of row-wise) to overcome the "Memory Wall" [5, 6]. This term was coined when database researchers noted that approximately half of database query execution time was spent on memory stalls during read operations [4] - the CPU was being starved of data and this was due to the inadequacies of the traditional row-store layout. Column-store layouts have demonstrated an ability to outperform row-store layouts by an order of magnitude (or more) during read operations due to their cache-friendly data structures, which minimize memory stalls.

The relational join, being one of the most important database operators, can also be one of the most processor and memory resource intensive operations. A popular state-of-the-art approach to performing relational join operations on main-memory column-store databases is through the use of the Radix-Clustered Hash Join [6], due to its ability to overcome memory access stalls by using cache-friendly data structures. However, existing join algorithms make an assumption that there is an unlimited reserve of temporary memory available, which can prove problematic in situations where there is insufficient free temporary memory to fully partition the input relations of the join operation. For example, the Radix-Clustered Hash Join fully partitions the input relations before executing the join phase, therefore needing temporary memory at least the size of the input relations themselves. To compound this situation, multiple joins can be performed at the same time in a pipelined manner, and thereby requiring their aggregate temporary memory space. When main memory is exhausted the virtual memory system pages data to disk, significantly lowering the performance of the join operation.

An existing join algorithm that requires virtually no temporary memory resources is the Nested Loops Join, which compares every tuple in the outer relation with every tuple in the inner relation. However, contrary to the Radix-Clustered Hash Join, the Nested Loops Join does not attempt to prune the number of comparisons, and therefore

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

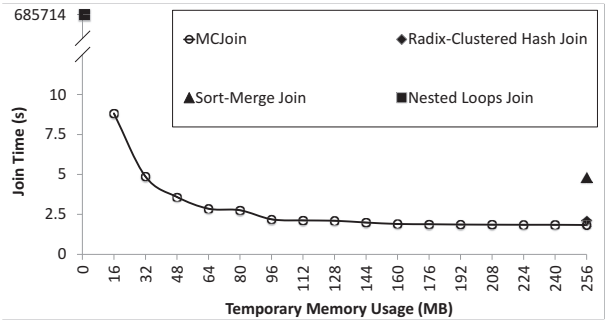


Figure 1: Join algorithm temporary memory usage and performance comparison.

typically gives poor performance. Ideally, we desire a join algorithm that can offer the performance benefits of a Radix-Clustered Hash Join, but with the ability to constrain the temporary memory overheads as needed, even to a level approaching that of the Nested Loops Join.

Figure 1 shows the performance achieved by the Memory Constrained Join (MCJoin) proposed by this paper. It demonstrates that MCJoin can achieve performance similar to the ideal mentioned above, namely being able to largely retain the performance of the Radix-Clustered Hash Join with very tight memory constraints. Even at a very small limit of 16MB (approx. 6% of the total memory used by the input relations), MCJoin outperforms the Nested Loops Join by six orders of magnitude. Note that the particular Sort-Merge Join and Radix-Clustered Hash Join algorithm implementations used for this comparison were by He et al. [10]. The temporary memory usage by the Sort-Merge Join algorithm was derived based on the fact the two relations need to be sorted out-of-place. In the experiment both relations had 16M tuples, and each tuple is a key/value pair totaling 8 bytes in size.

A simple and intuitive way to impose a memory limit on the Radix-Clustered Hash Join is to process the input relations in chunks (consecutive groups of tuples). However, this approach leads to an increased total number of partitioning and comparison operations because one pass through the entire inner relation is required for each chunk of the outer relation.

Our aim is to process larger chunks of the outer relation at one time while still obeying a memory limitation and retaining the performance benefits of a Radix-Clustered Hash Join. We have achieved this by designing and implementing the Memory Constrained Join (MCJoin) algorithm, a variation of the Radix-Clustered Hash Join. The results show that in a memory constrained environment, MCJoin can outperform a naive memory constrained version of the Radix-Clustered Hash Join algorithm (pioneered by Boncz et al. [6]) in all situations tested by margins of up to 500%.

MCJoin achieves this outstanding result by introducing *flexible*, *lightweight*, *lossless* data compression, memory constrained multi-level bit radix clustering, and a multi-purpose histogram.

MCJoin uses these features to achieve our objectives, as follows. Firstly, during the partitioning phase, MCJoin uses compression to reduce the size of partitioned tuples of the outer relation and therefore allowing larger chunks of the outer relation to be stored in temporary memory. This leads to fewer passes through the inner relation, thereby reducing the overall join cost. The compression algorithm exploits the nature of the hash partition process itself to get the compression and decompression almost for free.

Secondly, MCJoin restricts the size of the temporary memory used during radix clustering to a minimum while sacrificing almost no performance penalty, and reuses this memory space in an efficient manner throughout the join process. The extra space saved allows larger chunks of the outer relations to be processed at a time, thereby reducing the overall join cost.

Finally, much like other state-of-the-art Hash Join algorithms [10, 13], MCJoin employs a histogram to manage its partitioning needs. MCJoin improves upon the efficiency of this histogram by allocating the memory just once during the join process, and reusing the space allocated for the histogram for multiple purposes.

In summary this paper makes the following key contributions:

1. Identifies the importance of making join algorithms for main-memory databases memory constrained. This is in contrast to all existing literature which assumes unlimited available temporary memory.
2. Proposes a highly efficient memory constrained join algorithm called MCJoin. The algorithm has the following features: lightweight, lossless compression; high-speed multi-level radix clustering; and a multi-purpose histogram.
3. Finally, a very detailed empirical study of the performance of MCJoin versus a naive memory constrained version of the highly competitive state-of-the art Radix-Clustered Hash Join algorithm was conducted. The results conclusively show the superiority of MCJoin in a variety of situations.

The rest of this paper is organized as follows: Section 2 examines the related works in the field of main-memory database joins, focusing on column-based storage and hash joins. Section 3 presents our problem definition. Section 4 analyzes a naive solution to our problem. Section 5 presents the MCJoin algorithm. Section 6 evaluates the performance of MCJoin and contrasts it against an implementation of a Radix-Clustered Hash Join. Section 7 concludes and provides direction for future work.

2. RELATED WORKS

In this section we will present existing work in the area of column-based main memory relational databases. Our particular areas of focus will be on column-stores, join algorithms for column-stores, and data compression for column-stores.

2.1 Column-Store Databases

Since their inception in 1970 [8], relational database management systems have undergone extensive research to optimize performance. Much of this research has been focused on minimizing disk I/O latencies and stalls due to memory hierarchy. However, researchers in the late 1990s found that memory stalls were becoming a bigger bottleneck than disk I/O.

Ailamaki et al. [4] showed that approximately half of query execution time was spent on memory stalls, and furthermore, that 90% of these stalls were due to data misses in level 2 cache. Boncz et al. [6] proposed the use of column-store data layouts (i.e. decomposition storage model [9, 12]) and cache-aware algorithms to overcome these shortcomings, and developed MonetDB, an open-source DBMS particularly well suited for main-memory databases and displaying performance an order of magnitude greater than that of contemporary systems.

Empirical evidence has shown that column-store databases can offer performance advantages in orders of magnitude over traditional row-store databases in read-only scenarios, such as OLAP and querying scientific databases [2, 18]. The idea behind column-wise storage is that queries which use

only a very few columns from a table can load just the columns needed and therefore avoid loading irrelevant data from non-sought columns. This results is much more efficient CPU cache usage. Based on a similar design philosophy, Stonebraker et al. [18] developed C-Store, which added a number of features to column-stores such as various types of compression, storing overlapping column-oriented projections, etc. Zukowski et al. [14] developed a new execution engine for MonetDB, named MonetDB/X100. The performance of MonetDB/X100 was an order of magnitude better than even MonetDB. This was achieved mainly through a highly efficient use of compression in the system so that data can fit better in the CPU caches and thereby reduce the number of times data needs to be fetched from main memory. Numerous other papers [1, 2, 3, 11] on column-store databases have improved and explored the performance trade-off of various aspects of column-store databases.

2.2 Join Algorithms for Column-Stores

We first review the work by Boncz et al. [6] who developed the state-of-the-art Radix-Clustered Hash Join algorithm for column-store databases in Section 2.2.1. We devote a large section on this work since it forms the foundation of the join algorithm proposed in this paper. Next we review more recent work in the area by Kim et al. [13] in Section 2.2.2.

2.2.1 Radix Clustering

During the research of the MonetDB project, the authors referenced the design behind a main-memory implementation of the Grace Join by Shatdal et al. [17], which had a partitioning system designed to be level 2 cache friendly. Analyzing in detail the memory access patterns of the join algorithm on a variety of host systems, the authors noted that in some scenarios, the cost of a Translation Lookaside Buffer (or TLB) miss was more costly than that of a level 2 cache miss.

The TLB is used to speedup the translation of virtual memory addresses to physical memory addresses. The TLB acts as a kind of cache for these translations, keeping a small buffer of recent memory page translations (on the Intel i7 processor this buffer is in two levels, containing 64 addresses in the first level and 512 at the second level). Whenever a request to a logical memory address is made, the TLB is checked to see if this translation has already been cached. If the translation is present, the physical address is retrieved at very low cost. If it is not present, then the Memory Management Unit must resolve this translation, which can be costly (especially if many repeated requests are made to disjoint memory addresses).

Recognizing the importance of the role played by the TLB within a hash join scenario, the MonetDB team proposed the *Radix-Cluster Algorithm* [6, 15, 5]. This algorithm describes a TLB-friendly means of partitioning data prior to a hash join. In its simplest form, the algorithm seeks to partition the input relations into H clusters. For each tuple in a relation, it considers the lower B bits representing an integer hash value, and inserts that tuple into the corresponding cluster with a hash value matching that described by the lower B bits of the key.

Two detrimental effects may occur if H grows too large. Firstly, if H exceeds the size of the TLB cache, then each memory access into a cluster becomes a *TLB miss*. Secondly, if H exceeds the available cache lines of level 1 or level 2 cache, then *cache thrashing* occurs, forcing the cache to become polluted and the number of cache misses to rise.

The strength of the *Radix-Cluster Algorithm* is its ability to perform clustering over multiple passes. The lower B bits of the hash key can be clustered in P passes, where each pass

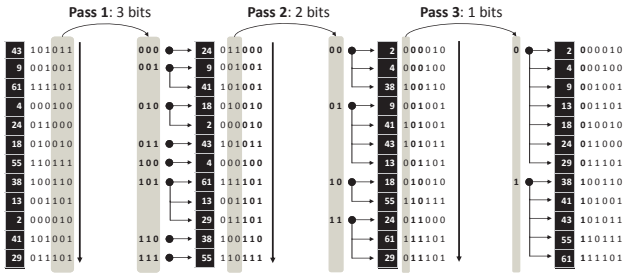


Figure 2: Radix Clustering Example

clusters the tuples on B_P bits, starting with the lowest bits (i.e. the right-most bits on little-endian CPUs such as the Intel x86 and x64 series), such that $\sum_1^P B_P = B$. For each pass, the algorithm subdivides every cluster into $H_P = 2^{B_P}$ new clusters, giving a total $H = \prod_1^P H_P$ clusters.

By controlling the size of B_P each pass, the number of clusters (H_P) generated can be kept to be under the count of TLB entries and cache lines available on that CPU, and therefore completely avoid the *cache miss* and *TLB miss* penalties.

Figure 2 demonstrates the hash values of tuples in a relation being clustered in three passes. Note that in this example, we represent our values with six bits, and the right-most bit is the least-significant bit. We will consider all six bits ($B = 6$) for clustering - however, in practice we would not cluster using all bits (e.g. we may only cluster on three bits, stopping after pass 1). In the first pass, $B_P = 3$, producing $2^{B_P} = 2^3 = 8$ clusters. Therefore, for each tuple, the lowest three bits are used to determine which cluster that tuple belongs to. In the second pass, $B_P = 2$ producing $2^{B_P} = 2^2 = 4$ sub-clusters for every cluster produced in the first pass. For each cluster, every tuple is placed into the corresponding sub-cluster based on the two bits starting from the fourth bit from the right. The third pass clusters on the last remaining bit, so $B_P = 1$ producing $2^{B_P} = 2^1 = 2$ sub-clusters. Our tuples are now clustered according to a six bit hash value.

The *Radix-Cluster Algorithm* demonstrates an extremely fast method of partitioning data by observing and working within the technical limitations of the host CPU. However, one downside of the algorithm is that there can be a significant memory overhead cost. The clustering is not performed "in-place" for the input relation, and therefore an entire copy of the whole relation is required for partitioning, with additional memory overheads per clustering pass (subject to implementation). Furthermore, an examination of the radix clustering algorithm implementation provided by Boncz et al. shows that their implementation dynamically allocates memory for each cluster on an as-needed basis. Repeated requests for memory allocation and de-allocation can accumulate into a significant performance overhead, and can also lead to memory fragmentation. In contrast, MCJoin pre-allocates all memory required for each radix cluster pass and thus does not incur the overhead of dynamic memory allocation.

2.2.2 Sort vs. Hash Join, and Histograms

When conducting research to create the fastest published join implementation for a main-memory database, Kim et al. [13] compared the performance characteristics of the Sort-Merge Join and Hash Join. Both implementations were heavily optimized for the target platform (Intel i7 965), featuring hand-crafted SSE and assembly language tweaks to fine tune the performance of the algorithms. Their empirical results indicated that the popular Hash Join algorithm still held the absolute performance crown (being 2× faster over

128 million tuples), but the Sort-Merge Join was gaining ground rapidly and it was suggested that features contained in the next-generation Intel CPUs may tip the favor towards Sort-Merge.

The Hash Join implementation described by Kim et al. contained some notable features. It shared much in common with the overall design philosophy of MonetDB, being built around a column-store layout and featuring tuples consisting of two 32-bit key / value pairs. During the partitioning phase of the Hash Join implementation, Kim et al. used a layered Radix-Cluster algorithm that was functionally similar in intent to that described by Boncz et al. [6]. However, rather than dynamically allocating memory to partitions on an "as-needed" basis, Kim et al. instead pre-scanned the input relations and built a histogram of the partition contents. Using this method, the authors were able to pre-allocate the destination memory for the partitioning, avoiding the overheads of repeated memory allocation requests and memory fragmentation. A prefix sum scan of the histogram would yield the starting offsets of each partition from the base address of the pre-allocated memory.

2.3 Compression Algorithms for Column-Stores

There are many compression techniques [1, 14, 18] used in column-store databases. The aim of the compression is to save storage space and also improve performance by reducing I/O costs and CPU cache misses. The compression techniques used by these works are typically lightweight in terms of computation. The techniques include run-length encoding, dictionary encoding, delta (or differential) encoding, bit-vector encoding, etc. These compression techniques differ from the compression algorithm used in MCJoin in two respects. First, MCJoin exploits the nature of the hash partition process itself to get the compression and decompression almost for free. Second, MCJoin does not assume the input relations are stored in compressed form but rather only compresses/decompresses the input data in temporary memory during the processing of the join itself. MCJoin can be used in conjunction with existing compression schemes which work with existing hash join algorithms. Note, compression techniques such as run-length encoding and delta encoding, in which later values depend on earlier values, will not work in compressed format with existing hash join algorithms. In contrast, compression schemes that do not have dependencies between values, such as dictionary encoding, will work.

3. PROBLEM DEFINITION

In a main-memory database scenario, we seek to perform a relational join operation while minimizing computation time and obeying a memory constraint. We assume the source data is stored column-wise, consisting of tuples closely styled after the Binary Units (or BUNs) of MonetDB [6] (which was also used by [13]), and that the source columns are currently held in entirety in main-memory. In this paper we work with tuples consisting of a 32-bit *key* attribute, and a 32-bit *payload* attribute. The key attribute represent the join key. The payload attribute is an index into a row of a relation. This index can be used to retrieve the value of one or more columns of the relation. 32 bits is enough to represent both the key and payload attributes because typically there are less than 2^{32} rows in a relation which in turn means there is less than 2^{32} unique values in a column.

Given two input relations R and S, we wish to perform an equijoin where R is the *outer* relation and S is the *inner* relation, $R \bowtie_{R.key=S.key} S$.

4. ANALYSIS OF A NAIVE MEMORY CONSTRAINED JOIN

In this section we first analyze the memory usage of a non-memory constrained Radix-Clustered Hash Join algorithm. Next, a naive memory constrained version of the join algorithm is analyzed. Finally we briefly outline how MCJoin improves upon this naive join.

In a main-memory database scenario, a conventional Radix-Clustered Hash Join algorithms fully partitions the input relations into temporary radix buffers (TRB). In this case the temporary memory usage equals the size of the entire input relations. Furthermore, if the radix clustering is performed over multiple stages (e.g. the algorithm is translation look-aside buffer (TLB) aware), we may find peak temporary memory usage increasing with each stage. Such a scenario is shown in Figure 3a.

A straightforward approach to constraining the memory usage is to only partition a consecutive group of tuples (or "chunk") of relation R at a time, and thus iterate over the entirety of R on a chunk-by-chunk basis. This approach is demonstrated in Figure 3b, where relation R is divided into n equally-sized chunks (R_C), such that $\sum_{i=1}^{n} |R_{C_i}| = |R|$. By processing R in chunks, we can use a smaller buffer for the radix cluster partitioning. Using this approach, we find that the memory required to host each TRB is the same as that utilized by R_C, and therefore the peak temporary memory usage is now twice R_C (i.e. $2 \times R_C$ instead of $2 \times R$).

Using a chunking approach, whilst more economical in its memory usage, does come at an increased processing cost. This is due to the the fact that the entire relation S needs to be partitioned per chunk of R. Therefore, the more chunks that R is broken up into, the more times S needs to be partitioned. This can be seen in the following simplified equation of joining R and S:

$$C_{Total} = C_{Part}(R) + \left\lceil \frac{|R|}{|R_C|} \right\rceil \times C_{Part}(S) + \left\lceil \frac{|R|}{|R_C|} \right\rceil \times C_{Join}(S, R_C) \quad (1)$$

Where C_{Total} is the total processing cost, $C_{Part}(x)$ is the processing cost to partition relation x, $C_{Join}(x, y)$ is the processing cost to perform a join operation between all tuples in relations x and y, $|R|$ is the cardinality of relation R, $|S|$ is the cardinality of relation S, and $|R_C|$ is the cardinality of relation chunk R_C.

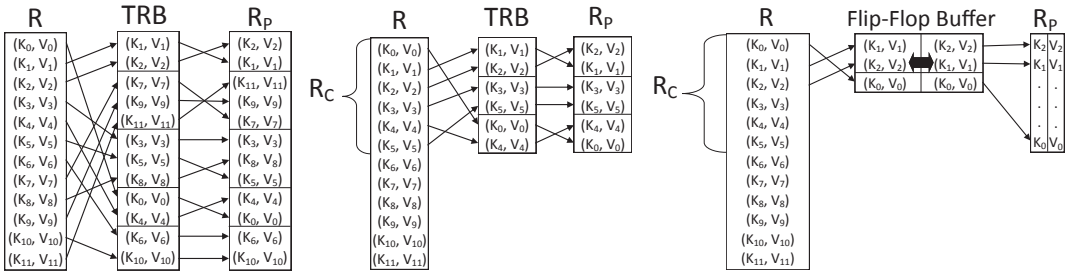
Equation 1 is the sum of the costs of three the major MCJoin processing phases: (i) the cost of partitioning R; (ii) the cost of partitioning S; and (iii) the cost of joining the chunks of R with S. Hence, Formula 1 can be stated as:

C_{Total} = cost of (i) + cost of (ii) + cost of (iii).

As R is the outer relation of the join, we only incur the partitioning cost once. Thus, the cost of (i) is just $C_{Part}(R)$. We are processing R in chunks of R_C. For each R_C of R, we partition all of S. Therefore, the cost of (ii) is $\left\lceil \frac{|R|}{|R_C|} \right\rceil \times C_{Part}(S)$ (where $\left\lceil \frac{|R|}{|R_C|} \right\rceil$ is the total number of chunks of R).

For each chunk R_C of R, we join all the tuples in R_C with those of S. Thus, the cost of (iii) is $\left\lceil \frac{|R|}{|R_C|} \right\rceil \times C_{Join}(S, R_C)$.

Equation 1 assumes that $|R_C|$ is constant, and that the size of the chunks of S has negligible impact on performance (this assumption is made on the basis that partitioning costs are calculated per-tuple with no additional overheads). Once given the join algorithm and the input relations R and S, the only part of Equation 1 that is variable is $|R_C|$. From the equation we can see that a larger $|R_C|$ results in lower total processing cost. However, in a situation where memory is constrained maximizing $|R_C|$ is a non-trivial task.



(a) Conventional Radix Clustering (b) Constrained Radix Clustering (c) MCJoin Partitioning

Figure 3: Partitioning examples

MCJoin maximizes $|R_C|$, by restricting the TRB to a small fixed size and minimizing the size of the partitioned data via compression. A simplified overview of the MCJoin partitioning process is shown in Figure 3c. Firstly, we can see the TRB, which we refer to as a *Flip-Flop buffer*, is restricted in size. The Flip-Flop buffer (named due to the way it flips data between static buffers during clustering - see Section 5.3 for more information) may have a cardinality much lower than $|R_C|$. In such a case, the Flip-Flop buffer iterates over R_C piece-by-piece (without performance penalty) until all of R_C has been scattered into our partitioned buffer (right part of Figure 3c), R_P. The data in the partitioned buffer is stored in compressed form.

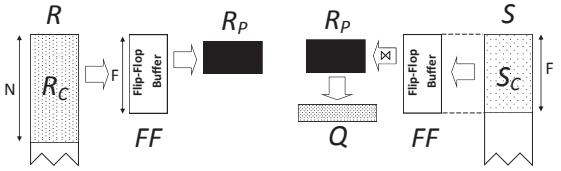
5. MCJOIN

We start by giving an overview of MCJoin, describing the key stages performed. Later in this section we describe each stage in more detail.

MCJoin is a variation of a Radix-Clustered Hash Join, where the input relations R and S are partitioned into "hash-buckets" (via high-performance Radix Clustering) to reduce the number of join comparisons made and therefore minimizing computational costs. MCJoin operates in two major phases: the Partitioning phase, and the Join phase. The Partitioning phase is responsible for partitioning tuples from relation R and storing them in a compressed format within Packed Partition Memory (R_P). The Join phase probes for join candidates within R_P for each tuple in S, and stores join matches in output set Q. As stated earlier in Section 4, in order to cope with a tight memory constraint we need to join relations R and S by parts. MCJoin does this by chunking R into smaller segments (R_C) and therefore restricting the memory needed for R_P (as shown in Figure 4a). To probe for join candidates in R_P, we take a similar chunking approach with relation S, subdividing it into smaller chunks (S_C) (see Figure 4b).

MCJoin introduces a "Flip-Flop Buffer", a high-performance, double-buffered, statically-allocated Radix Clustering mechanism that uses a fixed amount of memory. This Flip-Flop buffer is an important facet in MCJoin's approach to operating in a constrained memory environment, and is used in both the Partitioning and Joining phases. The Flip-Flop buffer is explained in further detail in Section 5.3.

Algorithm 1 shows a high-level overview of the operations of MCJoin. We define a number of functions, as follows. *DynamicallyAllocateMemory* allocates the constrained memory (denoted as M) between competing resources of MCJoin in such a manner that ensures fastest overall join execution time (see Section 5.4). *BuildHistogram* iterates over the current chunk R_C of relation R and builds a histogram of hash keys. This histogram is used for a number of purposes during both the partition and join phases (see Section 5.5). *Flip-Flop* radix clusters the input relations (see Section 5.3).



(a) Partitioning Phase (b) Joining Phase

Figure 4: Partitioning and Joining phases of MCJoin

PartitionUsingFlipFlopAndHistogram utilizes the histogram and Flip-Flop buffer to scatter the current chunk R_C into Packed Partition Memory, R_P (see Section 5.2). *Join* invokes the join probing procedure between S'_C (i.e. the radix clustered representation of S_C) and R_P (see Section 5.6).

Algorithm 1: High-Level MCJoin Algorithm

```
   input  : M = Memory Constraint, in bytes.
   input  : B = Hash Bit Length.
   input  : R = Input relation (outer).
   input  : S = Input relation (inner).
   output: Q = output set containing joined tuples of R and S.
 1 begin
 2     DynamicallyAllocateMemory (M, B);
 3     foreach R_C in R do
 4         H ← BuildHistogram(R_C);
 5         R_P ← PartitionUsingFlipFlopAndHistogram(R_C, H);
 6         foreach S_C in S do
 7             S'_C ← FlipFlop(S_C);     // S'_C is radix clustered version of S_C
 8             Q ← Q ∪ Join(S'_C, R_P);
 9         end
10     end
11 end
```

5.1 MCJoin Hash Key and Partitioning

As described in the overview, MCJoin operates in a similar manner to the conventional Hash Join, where tuples from input relations are scattered (based on a hash function applied to a key attribute) into hash table partitions. The matching hash partitions of each of the input relations would then be probed for join candidates. In a manner similar to most existing Radix-Clustered Hash Joins[6, 10, 13], MCJoin uses the least-significant B bits of the key attribute in the tuple as a hash key. This simple technique has two advantages - firstly, it is fast, and secondly it forms the basis of our compression technique described in Section 5.2.1. The high performance is possible due to the use of a singular bitwise AND operation (with an appropriate bitmask) to calculate the hash key. We also find that as each hash bucket now contains items with identical least-significant B bits (matching the hash bucket key), we are able to remove this redundant data from the stored items, forming the basis of our compression technique.

While this approach may be considered susceptible to data skew, we evaluate MCJoin against skewed data in Section 6.2 and have found that there was no performance degradation due to using this hash function.

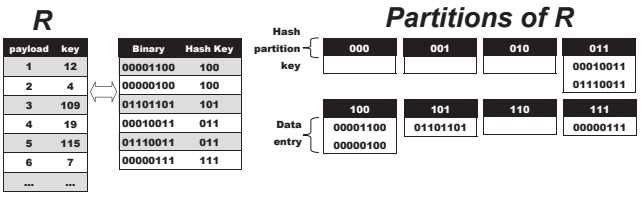


Figure 5: Simplified MCJoin partitioning example

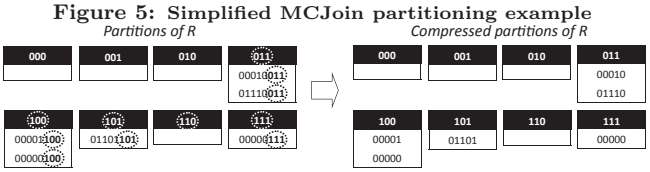


Figure 6: Compressed partitioning example

The number of hash partitions produced from this hash function is 2^B. An example of the simplified MCJoin partitioning system is shown in Figure 5. For simplicity, we assume that the tuple attributes in R are only 8-bit values, and we set $B = 3$. Therefore, the three least significant bits of the hash key (in this case, the value attribute) are used to determine what partition the value scatters into (note that we assume little endian architecture). As $B = 3$, we have $2^B = 2^3 = 8$ partitions for our hash table.

5.2 Data Compression

So far, we have examined how MCJoin builds the hash partitions by treating the compression algorithm as a black box. In this section, we will describe the MCJoin compression algorithm.

There are two kinds of compression utilized by MCJoin - *hash key compression*, and *payload compression*. Both are utilized simultaneously, but operate on different tuple attributes. The hash key compression system is applied to the join attribute. For the example shown in Figure 5, we are using *R.key*. Payload compression applies to the non-join attribute that will be needed to produce part of a joined output tuple. From Figure 5, this would be *R.payload*.

Regardless of the compression employed, we shall refer to the region of compressed memory holding the hash partitions as "Packed Partition Memory".

5.2.1 Hash Key Compression

Our first stage of data compression removes redundant data from the hash key. As demonstrated in Figure 6, the right-most (i.e. least significant) B bits of every data entry in each hash partition is redundant (as it is the same as the hash partition key). By stripping off the redundant bits of each item stored in the hash table, we are able to express the input data while using less bits than the original relations. In the example shown in Figure 6, we are able to represent an 8-bit value using only 5-bits. This represents a memory savings of $\sim 38\%$ compared to a conventional hash bucketing method.

Whilst we are able to calculate a shortened bit-wise representation of the data entries, it is not a trivial matter to simply store these values verbatim into RAM. Instead, we must write values into memory at a wider grain - typically in words of 8, 16, 32, or 64 bits. Normally, this would be done at the processor's native word bit-width (our experimental system uses a word length of 64-bits). To take advantage of our condensed representation of values, we must sequentially bit-pack these shortened values into native-width words. We pack these shortened bits into native words through a combination of bitwise SHIFT and OR operations.

It is possible that a condensed value will "stride" two native-width words in memory. This will occur when B does not cleanly divide into the native-word format. To deal with such situations, two memory reads and two memory writes will be required.

Using the Histogram (described in Section 5.5), we are able to convert the ordinal position for any item of interest within Packed Partition Memory into a word and bit offset for retrieval.

Performance Analysis. Given the number of bit manipulations required to insert a value into packed partition memory, it may initially appear that the memory writing performance may be slower than a traditional Hash Join method of partitioning, which requires no bit manipulation for storage. However, an analysis of MCJoin's compression based partitioning strategy reveals that the extra costs of bitwise operations to store compressed data is largely offset by the cache-friendly nature of the data access patterns, and the low CPU-cycle costs of bitwise operators. This results in only a slightly higher overall partitioning time for MCJoin.

When accessing RAM, modern CPUs read a cluster of memory around the item of interest. A copy of this cluster (or more correctly, a "cache line") is kept in local cache memory. Subsequent requests to nearby memory addresses can be retrieved from the fast cache memory instead of slower RAM. The Intel i7 has a cache line width of 64 bytes, and as such read requests to RAM will retrieve 64 bytes of data localized around the address of interest. Therefore, in ideal circumstances, we can get 8×64-bit values from the cache line with one RAM access. By packing more data into this space, we maximize the amount of useful data transferred per memory request.

Furthermore, it is far less expensive (in terms of CPU cycles) to shift and mask out bits of interest from cache memory when compared to the cost of reading from RAM. On the Intel i7 CPU, the bit-shifting operators (SHL / SHR) and the bitwise AND instructions have a throughput of 0.5 and 0.33 respectively. That is, we can have two bit-shift or three bitwise AND operations running concurrently per clock cycle. This compares favorably against memory transfers from RAM, which may take hundreds of clock cycles to complete.

5.2.2 Payload Compression

By removing redundant information from the key attributes within packed partition memory, we were able to realize savings in storage space for the key attributes. However, the same technique is not possible for the payload attribute, as the contents of these attributes bears no commonality with the container partitions themselves.

One naive technique to store the payload attribute would be to simply include this data along with the compressed key attributes in a manner somewhat similar to how a conventional Hash Join would operate. However, we can see two immediate shortcomings from this technique:

1. The naive technique not saving any storage, and end up polluting the contiguous packed partition memory with data unrelated to the key attribute. This results in less efficient bandwidth utilization.
2. Given a join with low selectivity, much of this payload data will remain unused, and therefore consumes memory for no immediate gain.

Therefore, rather than store the payload data directly into packed partition memory, we will instead store an offset vector to the parent tuple. This vector will act as a relative offset from the base address of the current chunk of the input relation. The bit width of each offset vector entry can be minimized based on $|R_C|$.

Figure 7 demonstrates the offset vector calculation. For a given input chunk R_C, we determine the base address to the first tuple of R_C. Therefore, for any given i^{th} tuple R_C, we derive its offset to be i. The minimum number of bits needed

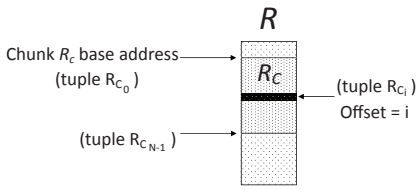


Figure 7: MCJoin Offset Vector

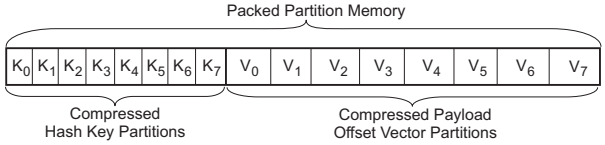


Figure 8: Packed Partition Memory

to represent the offset value for $|R_C|$ tuples is $\lceil \log_2(|R_C|) \rceil$ bits.

With the offset vector stored in a compressed format, we can use a bit shifting system identical to that used by the hash key to store the offset into a separate packed partition memory space allocated for offset vectors only.

Therefore, we have two separate packed partition memory spaces: one for compressed keys, and another for compressed payload offset vectors (see Figure 8). Matching key and offset vectors can be found in the corresponding ordinal positions in each memory space. Partition starting locations, and item ordinal positions, can be calculated via the Histogram as described in Section 5.5. The offset vectors can be utilized to re-stitch matched candidate tuples during the joining phase of the MCJoin algorithm - this is described in Section 5.6.

5.3 Flip-Flop Buffer

As mentioned earlier MCJoin takes the radix clustering approach to partition relations in order to reduce the performance penalties associated with random memory access. However, unlike existing work, MCJoin does not continuously dynamically allocate memory to perform the clustering, but instead pre-allocates up to two buffers (for situations where two or more passes are required to complete radix clustering) of a pre-determined size, and "flips" the data being clustered between these two buffers (hence the name, *Flip-Flop Buffer*). The pre-allocation of the buffers avoids the costs of dynamic memory allocation, and to further increase this efficiency, the Flip-Flop Buffer is only created once and then reused throughout the MCJoin process. We allow the Flip-Flop buffer to be of any size between $|R_C|$ and $\frac{|R_C|}{32}$ (where 32 is a value particular to our experimental system - please see Section 5.4 for its relevance). This is because we break up R_C into smaller subdivisions and radix cluster each subdivision separately.

Determining the correct size of the Flip-Flop buffers is of critical importance because Equation 1 in Section 4 implies that significant processing cost savings come from maximizing $|R_C|$. As both R_C and the Flip-Flop buffer compete for the same limited memory resources, we aim to lower the capacity of the Flip-Flop buffer in order to maximize $|R_C|$. Taking this approach, a given chunk R_C of relation R may itself be processed by parts, with each part being individually radix clustered by the Flip-Flop buffer in turn.

This approach of lowering the capacity of the Flip-Flop buffer not only grants overall performance gains by maximizing $|R_C|$, it also grants higher Radix Clustering throughput when the Flip-Flop buffer capacities are of a size that can fit into the processor's cache memory. This is demonstrated in Figure 9, where our experimental system (as described in Section 6.1) is tasked with radix clustering a relation of 16 million tuples. We vary the size of the Flip-Flop buffer from

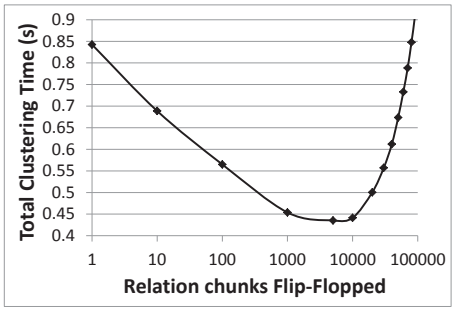


Figure 9: Flip-Flop Clustering Performance vs. Iterations

a capacity that can service the entire relation in one pass, down to a size that requires the relation to be divided into 100,000 parts for the Flip-Flop buffer to process. We find that smaller Flip-Flop buffers correlate to increased partitioning performance, with peak performance found at approximately the 1,000 iteration mark. At this size, we find that the Flip-Flop buffer fits entirely into the L2 cache of our i7 CPU (i.e. 256KB), yielding maximum performance. It is important to note that a Flip-Flop buffer that is too small can incur performance penalties, due to the execution overheads becoming a significant cost.

The results shown in Figure 9 imply that smaller Flip-Flop buffers do not increase clustering time, as long as care is taken not to choose a buffer size that is so small that overall performance will be penalized. Therefore, we can choose a small flip-flop buffer size in order to maximize $|R_C|$.

This same Flip-Flop buffer radix clusters chunks of both the outer (R_C) and inner relations (S_C).

Whilst our analysis of the Flip-Flop buffer shows that smaller-sized buffers are capable of yielding increased radix clustering performance, it does not take into account the amortized costs of invoking the Flip-Flop clustering mechanism during the execution of MCJoin. All invocations of the Flip-Flop buffer are followed by either a scatter operation (during the partitioning phase) or a join operation (during the join phase), which incur their own overheads. Therefore, choosing the best-sized Flip-Flop buffer based solely on the data presented in Figure 9 may not yield the expected results. Our computational cost model (see Equation 1) suggest that the dominant factor in join execution cost is $|R_C|$, and the analysis shown in Section 5.4 verifies this model. Our analysis of the Flip-Flop buffer's performance therefore demonstrates the reasons why it is acceptable for the Dynamic Memory Allocation system to choose smaller-sized Flip-Flop buffers while attempting to maximize $|R_C|$.

5.4 Dynamic Memory Allocation

Given a memory limit and a hash bit length, we describe how MCJoin allocates memory between its three competing data structures of histogram, packed partition memory for R_C and the Flip-Flop buffer. The size of the histogram is fixed, once the hash bit length is set. The key concern is how to allocate the remaining memory between the packed partition memory for R_C, and the Flip-Flop buffer. As mentioned in Section 4 larger $|R_C|$ results in lower total cost, because it results in less passes through the inner relation, as explained by Equation 1. We have developed a simple yet effective approach towards dynamic memory allocation.

The idea behind Algorithm 2 is that we first look at how many chunks we would partition R into if we assigned all of MC_{Remain} to the Packed Partition Memory. We do this using Lines 2 and 3 of the algorithm. R_{LBNC} gives us a lower bound on the number chunks of R for a given MC_{Remain}. Then in Line 3 we compute the number of tuples per chunk of R (R_{TPC}) if we evenly distributed the tuples in R into

Algorithm 2: Dynamic Memory Allocation

```
   input : MC_Remain, Memory Constraint after subtracting the memory
           used by the histogram, in bytes
   input : B, Hash Bit Length.
   output: M_PackedPartitionMemory, size of Packed Partition Memory,
           in bytes.
   output: M_FlipFlop, size of the Flip-Flop buffer, in bytes.
1  begin
       /* NumberOfPackedTuples returns the number of tuples that fits into Packed
          Partition Memory of a given size                                  */
2      R_MaxTuples ← NumberOfPackedTuples(MC_Remain, B);
3      R_LBNC ← ⌈ |R| / R_MaxTuples ⌉;
4      R_TPC ← ⌈ |R| / R_LBNC ⌉;
       /* GetPackedMemoryAllocated returns the Packed Partition Memory size in bytes
          that fits a given number of tuples                                */
5      M_PackedPartitionMemory ← GetPackedMemoryAllocated(R_TPC, B);
6      while M_PackedPartitionMemory < α do
7          R_LBNC ← R_LBNC + 1;
8          R_TPC ← ⌈ |R| / R_LBNC ⌉;
9          M_PackedPartitionMemory ← GetPackedMemoryAllocated(B, R_LB);
10     end
11     M_FlipFlop ← MC_Remain − M_PackedPartitionMemory;
12 end
```

R_{LBNC}. Note that $R_{TPC} \neq R_{MaxTuples}$ because of the ceiling function used in Lines 3 and 4. We use this difference between R_{TPC} and $R_{MaxTuples}$ to fit the Flip-Flop buffer. As mentioned in Section 5.3 the Flip-Flop buffer may actually perform better when it is small (up to a point). We introduce a parameter α that prevents the Flip-Flop buffer from being too small. In our experiments we found setting $\alpha = |R_{TPC}|/32$ works well. If $M_{PackedPartitionMemory}$ is below α we lower $|R_{TPC}|$ to be a lower multiple of $|R|$ until $M_{PackedPartitionMemory}$ becomes larger or equal to α. (Lines 6 - 10). Finally we assign the Flip-Flop buffer the portion of MC_{Remain} that is not used by $M_{PackedPartitionMemory}$ in Line 11.

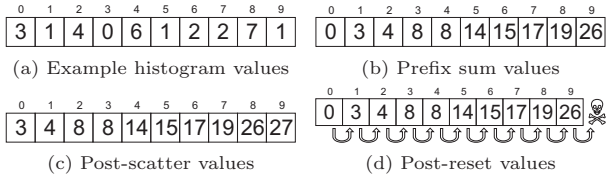
We evaluate the performance of our memory allocation algorithm by first measuring the performance under two scenarios: where we keep total memory limit M_{total} fixed and vary B (as shown in Figure 10a), and fix B and vary M_{total} (as shown in Figure 10b). For each combination of M_{total} and B, we manually search for the best memory allocation between Packed Partition Memory and the Flip-Flop buffer, by varying the ratio by 5% increments. We compare this performance range against our algorithm and note that our algorithm meets the best recorded time in all cases. The results show our dynamic memory allocation algorithm can find the best manually found memory allocation in all cases tested.

5.5 MCJoin Reusable Histogram

In this section we describe how MCJoin utilizes a histogram to manage partitioning, whilst being mindful about operating under a constrained memory environment. To avoid the overheads of dynamic memory allocation, MCJoin statically allocates a pre-defined region of memory just once to store a histogram, and then reuses this histogram to perform a number of different functions.

Specifically, MCJoin utilizes the histogram for two major phases of operation: partitioning, and join probing. During the partitioning phase, MCJoin requires the ability to determine the initial location of all partitions, and to track the current end of each partition as the partitions progressively fill up during the scatter process. During the join phase, MCJoin is able to use the histogram to find the number of join candidates for a given partition.

The life cycle of the MCJoin histogram is as follows. Prior to the partitioning phase, the capacity of the histogram is determined. The number of partitions utilized by MCJoin is directly related to B, and as we wish to have one 32-bit entry in the histogram for each partition, we find the total



(a) Example histogram values (b) Prefix sum values

(c) Post-scatter values (d) Post-reset values

Figure 11: Example histogram values, before and after prefix sum

capacity to be that of 2^B entries, and sufficient memory is therefore allocated to host the histogram.

We initially populate the histogram by counting the number of tuples in R_C that reside in each partition (see Figure 11a). Therefore each entry in Figure 11a represents the number of tuples residing in the corresponding partition of R_C.

Next, we perform a prefix sum upon the histogram values (see Figure 11b). Each entry of the histogram now represents the offset value (measured as a number of items from a base point in memory) of each partition.

After the prefix sum has been performed, we scan through R_C (via the Flip-Flop Buffer), and for each tuple encountered, determine the matching partition that it belongs to. We scatter this tuple to the packed partition memory address calculated from the offset contained in the corresponding partition entry in histogram, and increment the histogram entry. At any moment in time during this process the current histogram value represents the current end position of the corresponding partition of R_C. Once we have completed our sweep through R_C, the histogram will look like the example shown in Figure 11c.

After the scatter process, we need to reset the histogram to the original prefix-sum state because during the join phase we need to know where each partition of R_C starts. We need to effectively reset the histogram. This is performed by shifting every element into the successive position in the histogram, while discarding the last value and placing 0 into the first element (see Figure 11d). This simple and cost effective way of resetting the histogram works because at the end of the scatter process the end location of partition i is the same as the beginning of partition $i+1$, since the partitions are stored contiguously.

The histogram is maintained during the join probing phase of MCJoin. This is further described in Section 5.6.1.

To locate the J^{th} item from partition P in packed partition memory, we perform the steps as shown in Algorithm 3.

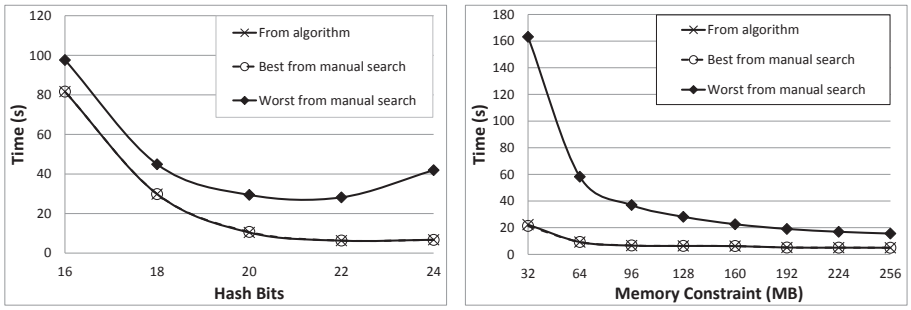
Algorithm 3: Locating items in Packed Partition Memory

```
   input : P = partition number.
   input : J = ordinal item number within position.
   output: Word = word offset from the beginning of packed partition
           memory (hash key, or vector offset).
   output: Offset = bit offset within word.
1  begin
2      Z ← GetHistogramValue(P);
3      Z ← Z + J;
4      Word = ⌊ (Z × C) / 64 ⌋;
5      Offset = (Z × C)  mod 64;
6  end
```

Where C is the number of bits written per entry. Note that as divisions are costly instructions for the CPU to perform, we can replace the division with a bitshift 6 positions to the right. Similarly, the mod instruction can be replaced by performing a bitwise AND with the value of 63.

5.6 Joining

Having completed the scattering and compression of a given chunk R_C of input relation R, the MCJoin algorithm will iterate over input relation S and probe for join candi-



(a) MCJoin Fixed 128MB Constraint (b) MCJoin Fixed 22-bit Hash

Figure 10: Dynamic Memory Allocation

dates in packed partition space R_P. As the tuples in S may be unordered, we incur the risk of random memory access patterns if we probe for join candidates on a "first come, first served" basis. Therefore, we wish to restructure the tuples in S to allow for cache-friendly memory access patterns.

We have already defined a "Flip-Flop" buffer during the partitioning phase, with a capacity of F tuples. The Flip-Flop utilizes our Multi-Level Radix Clustering technique to cluster tuples based on a key attribute of bit radix length B. We will utilize the input buffer to read one chunk S_C of S at a time, where the cardinality of S_C will be defined as F. We copy the chunk S_C into the input buffer, and perform radix clustering on the $S.key$ attribute using a bit radix length of B, producing the modified chunk S'_C. Once clustering is completed, we iterate over the clustered tuples of S'_C and probe for matches in R_P. The overall strategy is to progressively iterate through the chunks R_C of relation R as an outer loop and the chunks S_C of S as an inner loop. Given relations of unequal cardinality, the smaller relation would be represented as R as this would simplify maximizing $|R_C|$.

5.6.1 Probing Sub-Phase

The probing sub-phase is performed in two distinct steps. In step one, for a given tuple S'_{C_i}, we mask out the least-significant B bits of the key attribute (i.e. $S.key$) to determine which partition in R_P will contain join candidates. We then examine the histogram entry for this partition to see if there is a non-zero quantity of join candidates. Note that as the histogram contains prefix sum values, for a given partition P in histogram H, we find the number of join candidates X to be $X = H[P+1] - H[P]$. If $X = 0$ then we have no join candidates and move on to the next tuple $S'_{C_{i+1}}$. If $X \neq 0$ then we proceed to step two.

For step two, we calculate the beginning address of candidate partition P in R_P using the method described in Section 5.5. We then iterate over the X packed keys in R_P and continue probing for matches. Should a match be found, invoke the Join Matching Sub-Phase. Once we finish scanning all X items in R_P, we return to step one.

Performance Considerations. Our approach to probing favors the architecture of modern CPUs, such as the Intel i7, and exhibits memory access patterns that maximise data throughput. Due to S'_{C_i} being radix clustered, as we iterate over it linearly the probes into the histogram are also linear. As both S'_{C_i} and the histogram are stored in contiguous-memory data structures, we take full advantage of cache coherency and due to the restricted range of memory addresses being probed, we also gain the advantage of having full TLB backing.

5.6.2 Join Matching Sub-Phase

When a join match is located, we fetch the matching vector offset V (from the packed partition area for offsets) and write a tuple consisting of $(V, S.payload)$ into an output set Q. If Q has reached a predefined maximum cardinality of Q_{MAX} (i.e. the capacity of the pre-allocated output buffer) then we proceed to the Tuple Stitching Sub-Phase, otherwise we return to the previous phase.

Performance Considerations. To maintain high output performance, we pre-allocate a contiguous output buffer of a pre-defined size. Pre-allocating the output Q buffer avoids subsequent dynamic memory allocation requests being made to the operating system for each matched tuple. The contiguous memory layout also aids the tuple stitching sub-phase by being cache-friendly and encouraging read-ahead behavior in the CPU.

5.6.3 Tuple Stitching Sub-Phase

There are two conditions that can invoke the Tuple Stitching Sub-Phase - the cardinality of output set Q has reached a predefined value Q_{MAX}, or we have processed every tuple of set S'_C. If the cardinality of Q is non-zero, we iterate over every tuple of Q and read the offset vector, $Q.V$. By adding this vector to the base address of R_C, we can find the address of the source tuple R_i. We replace the value $Q.V$ with $R.payload$, finally yielding the joined tuple $(R.payload, S.payload)$. We save this result back into Q and continue processing the remaining tuples of Q in the same fashion.

Performance Considerations. We chose to implement the tuple stitching in batches (as a separate phase) for performance reasons. During the join matching sub-phase we seek to minimize working set size and maximize cache efficiency by observing locality of data. When a join match is found, if we immediately resolve the vector offset by fetching the corresponding tuple in R, we are likely to polute the CPU cache and thus have a detrimental effect on the overall performance of the join phase.

6. EXPERIMENTAL EVALUATION

In this section, we will conduct a series of experiments to compare MCJoin against a naive memory constrained version of the state-of-the-art MonetDB-styled Radix-Clustered Hash Join (RCHJoin) algorithm by He et al. [10]. Although the paper by He et al. is focused on joins using the graphics processing unit, they also provided a highly optimized implementation of a MonetDB-styled Radix-Clustered Hash Join. It is this implementation that we call RCHJoin.

We have modified RCHJoin in such a way that it is capable of running within a constrained memory environment. Firstly, we modified RCHJoin to process the input relations in chunks. Secondly, we used a dynamic memory allocation system similar to that described in Section 5.4 for RCHJoin. We verified that this dynamic memory allocation system allocated resources in a fashion that produced the fastest

join execution times, by using a testing procedure similar to those described in Section 5.4.

6.1 Experimental Setup

Our host platform is equipped with an Intel i7 860 CPU running at 2.8GHz on an Asus P7P55LX motherboard. It has 4GB of dual-channel DDR3 memory. The host platform runs under Windows 7 x64 Enterprise Edition. Our development environment is Microsoft Visual Studio 2010, using Intel C++ Compiler v12.0. The configuration setting for the compiler is "Maximize Speed plus High Level Optimizations". We restrict the database joining algorithms to run on a single processor core running at a fixed frequency. We use a single core since the focus of our work is on making join memory constrained instead of utilizing multiple cores. It should be possible to parallelize our implementation to take advantage of multiple cores like the existing work in this area[10, 13]. We leave this as future work.

Our primary focus when taking performance-based metrics is to measure the elapsed time a particular task takes to complete under experimental conditions. We will use the high-resolution performance counters found in most modern PCs for this purpose. All results reported represent an average of six sample runs.

Unless specified otherwise, the default parameters shown in Table 1 are used.

Parameter	Value
Cardinality of input relation R ($\|R\|$)	16000000
Cardinality of input relation S ($\|S\|$)	16000000
Column width (OID + value)	8 bytes
Number of hash partitions in bits B	24

Table 1: Default Parameters

We use the Boost C++ Libraries v1.44 [7] to generate randomized data. Specifically, we utilize the mt19937 generator, which is a specialization of the Mersenne Twister pseudo-random number generator [16]. We use a static random number seed for all experiments.

We generate both uniform and skewed distributed data. For uniform distribution we generate tuples of R and S which have key value uniformly distributed between 1 and $|R|$ and $|S|$, respectively. For skewed distribution we keep S uniformally distributed but the values in R are generated using a Gaussian distribution with a default sigma of 1 and mean of $|S|/2$. This means the values in R will match predominately on a small range of S values.

Controlling selectivity. For the uniformally distributed data we control selectivity by varying the fraction of tuples in R finding a match in S, we call this term β. This is done by keeping the value range of S fixed between 1 and $|S|$ and varying the value range of R between $|S|$ and $|S|/\beta$.

For skewed distributed data we again keep the range of S fixed and vary the range of R. When generating each value of R we first decide if it should match any tuple in S by generating a uniformally random number between 0 and 1 and comparing it against β. If we decide to not match anything in S we assign the tuple in R a value greater than $|S|$ (therefore outside the value range of S) otherwise we use the same skewed distribution method as explained above to pick a value within the value range of S.

6.2 Experimental Results

We conduct five experiments comparing MCJoin against RCHJoin. First, we compare the scalability of MCJoin against RHCJoin under no memory constraints. Second, we compare MCJoin against RCHJoin under varying memory constraints. Third, we compare MCJoin against RCHJoin while varying the selectivity of the data β. Fourth, we analyze the ability of MCJoin at handling data skew. Lastly, we analyze the performance of the join phase of MCJoin.

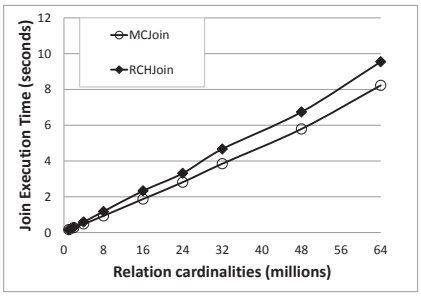


Figure 12: MCJoin vs. RCHJoin - Scalability Under No Memory Constraints

6.2.1 Scalability Under No Memory Constraints

We wish to establish a baseline performance benchmark for MCJoin in an unconstrained-memory environment. We varied the cardinality of input relations R and S to determine the scalability of MCJoin. For comparative purposes, we measured the performance of RCHJoin under the same scenario and contrast the performance against MCJoin.

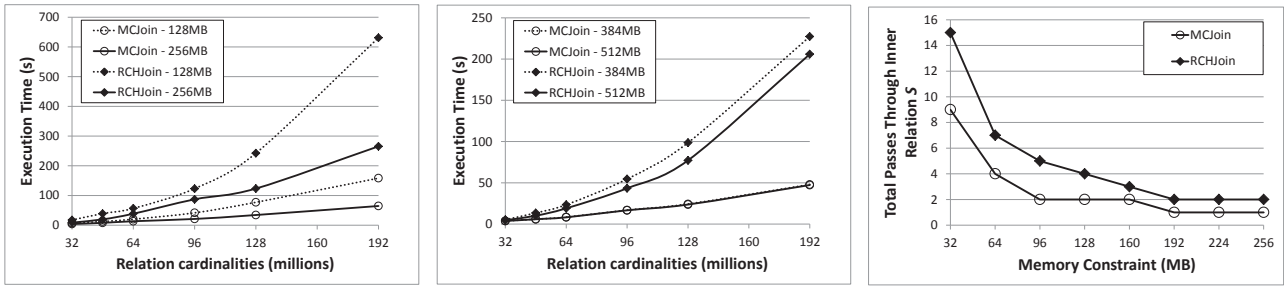
Due to the initial setup costs of the MCJoin algorithm, we expected that MCJoin would incur a small performance penalty to RCHJoin but as we were able to make better use of data compression for larger input relations, our expectation was that MCJoin will make up any lost ground to RCHJoin.

Our results (as shown in Figure 12) show that in an unconstrained memory environment, the overall performance of RCHJoin and MCJoin is quite similar, with a marginal advantage going to MCJoin. The elapsed time measurement scales quite linearly with relation size for both algorithms, suggesting efficient use of the available processing power and memory bandwidth. The results show MCJoin marginally outperforms RCHJoin despite the fact that MCJoin needs to perform compression whereas RCHJoin does not. The reason for this is two fold. First, MCJoin has a very efficient compression algorithm which incurs minimal compression and decompression overheads. Second, the multi-level radix clustering and multi-purpose histogram of MCJoin results in reduced CPU stall times due to memory latency.

6.2.2 Varying Memory Constraint

Figure 13 contrasts the performance levels of MCJoin and RCHJoin whilst operating in a constrained-memory environment. Both algorithms were granted the restricted memory spaces of 128MB, 256MB, 384MB, and 512MB, and for each restricted memory setting we varied the cardinality of the input relations. We increased these cardinalities up to a size where neither relation could be copied fully into the restricted memory space.

MCJoin's fast compression allows it to effectively get more tuples of R into R_C compared to RCHJoin, resulting in lower total passes through inner relation S (see Figure 13c). This performance difference is quite marked, as shown in Figure 13. Under the worst memory constraint MCJoin still outperforms the best times recorded for RCHJoin. For the majority of tests, MCJoin performed between $300\% - 400\%$ faster than RCHJoin. Of particular note for MCJoin is that the 384MB and 512MB constraint result times are almost identical. Even though more processing work was required for the 384MB test run, the results indicate that MCJoin had reached maximum saturation point for the available memory bandwidth of our experimental system.



(a) 128MB and 256MB Memory Constraints (b) 384MB and 512MB Memory Constraints (c) Total Passes Through Inner Relation S

Figure 13: MCJoin vs. RCHJoin - Constrained Memory

6.2.3 *Varying selectivity (β)*

In this experiment we vary the selectivity of the default data set under no memory constraints. We vary selectivity using the β term described in Section 6.1. β effectively represents the fraction of tuples in R matching at least one tuple in S.

The results shown in Figure 14a indicate that both MCJoin and RCHJoin respond in a similar manner (i.e. a gradual decrease in performance) to higher values of β when data is uniformly distributed. This matches our expectations, as both algorithms will find more join candidates in all hash partitions probed that must be resolved. However, close inspection of the results shown in Figure 14b suggest that MCJoin suffers a lesser penalty for higher values of β when the data is skewed. This would be a result of the join candidate probing method used by MCJoin, and we will take a closer examination of how this behaves in Section 6.2.4.

6.2.4 *MCJoin Skew Analysis*

The results shown in Section 14b give an indication that MCJoin exhibits a tolerance to skewed data. To examine this further, we set $\beta = 1$, and then we forced the skew to become progressively narrower by varying the σ parameter of the Gaussian distribution. We again conduct this experiment in a non-memory constrained environment.

We expected MCJoin to respond well to skew due to the way it probes for join candidates. As is shown by Algorithm 1 in Section 5, we iterate over the tuples in S'_C (the radix clustered copy of S_C) in a sequential manner and linearly probe the Histogram (H) for join candidates. This process is very cache friendly as all memory access is contiguous in nature, therefore being cache friendly and will also lend well to speculative execution on the CPU. In heavily skewed data, we will find that a very small subset of the tuples in S'_C find join candidates in a very small subset of H. When these probes continue into packed partition memory, the join phase will find all candidate matches linearly arranged in partition memory. This will allow for a very efficient resolution of joins.

The results shown in Figure 14c confirm our expectations. Whilst partitioning times gain little advantage from the increase in skew, the join phase finds such a data layout favorable and therefore takes less time to complete.

6.2.5 *MCJoin Join-Phase Analysis*

A Hash Join gains performance advantages over a Nested-Loops Join by partitioning the relational data in such a way that less comparisons are needed between the tuples of R and S to resolve the join. Therefore, the number of partitions used by the Hash Join should be directly related to the overall join performance. To examine this further, we varied the value of B, increasing the number of partitions used by MCJoin. We measured the time it took to scatter the data into these partitions, and also time it took for the joining phase to complete. We again conduct these experiments in a non memory constrained experiment. In these experiments we reduced the size of the data to $|R| = 1M$ and $|S| = 1M$ because for small values of B it took a long time to execute the default data set (starting to approach Nested-Loops Join performance).

Figure 14d quickly highlights the inadequacies of having too few partitions for MCJoin. Whilst not as slow as a Nested-Loops Join (our experimental system took 2213 seconds to resolve this join using Nested-Loops), a setting of $B = 4$ does not yield a timely join result when compared to higher values of B. For values of $B < 16$, the join phase accounts for the bulk of overall join processing time. For this smaller sized data a setting of $B = 20$ yields optimal results. While the join phase time continues to improve for larger values of B, the partitioning costs start to become significant and overall performance drops off.

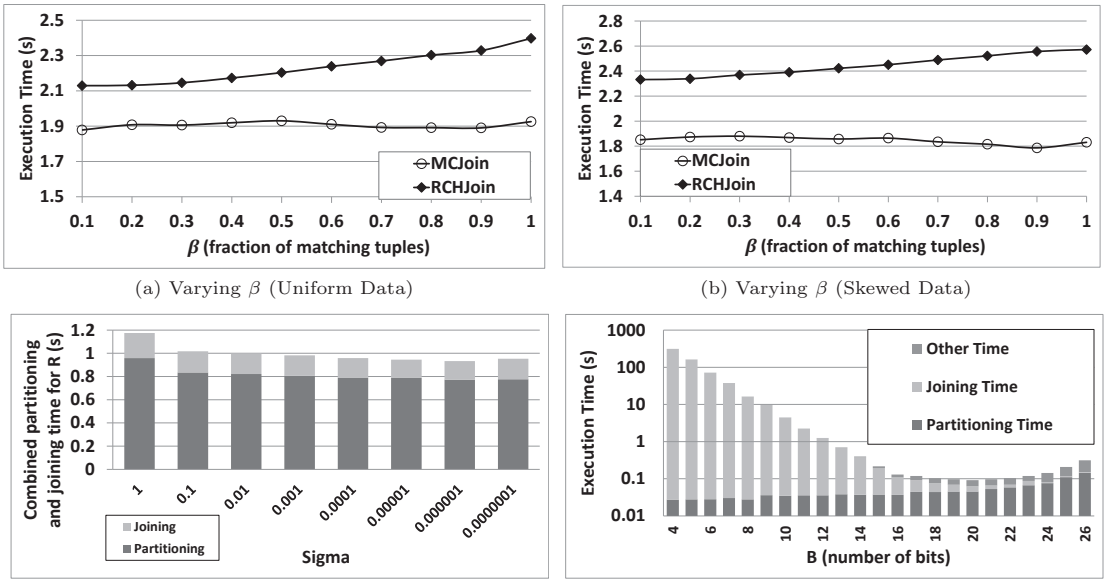
We expect that the partitioning costs will play a more significant role for relations of higher cardinalities, but we note as future work the ability to automatically tune B to the best value suited for given input relations.

7. CONCLUSIONS

The main contribution of this paper is to propose a memory constrained join algorithm called MCJoin. The algorithm uses a combination of lightweight, lossless compression, high-speed multi-level radix clustering, and a multi-purpose histogram to offer very high join performance in a memory constrained environment. This contrasts with all existing work in this area which assumes unlimited available temporary memory to perform the join.

Extensive experimental results show that MCJoin outperforms the a naive version of the current state-of-the-art Radix-Clustered Hash Join algorithm (RCHJoin) in a variety of situations. It is very encouraging to see that even in the unfavorable unconstrained memory environment MCJoin can still marginally outperform RCHJoin. This can be attributed to the highly-efficient compression algorithm of MCJoin and the efficient utilization of memory bandwidth by MCJoin. In a constrained memory environment, MCJoin can vastly outperform RCHJoin by almost 500%. MCJoin was shown to perform well under skewed data distributions.

Recently, Graphics Processing Units (GPU) have become easier to program for general purpose applications through such interfaces as NVIDIA's CUDA. The packed partition memory of MCJoin is particularly well suited for maximizing bandwidth efficiency of the bus between CPU RAM and GPU RAM. For future work we would like to pursue a GPU-enabled variant of MCJoin. Another direction of future work is to extend the MCJoin to handle multi-way joins. Finally, developing a memory constrained version of the Sort-Merge Join is another interesting direction of further work.



(a) Varying β (Uniform Data)

(b) Varying β (Skewed Data)

(c) MCJoin Skewed Analysis

(d) MCJoin Varying B

Figure 14: MCJoin vs. RCHJoin (Varying β), and MCJoin Analysis

8. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980, New York, NY, USA, 2008. ACM.

[3] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization strategies in a column-oriented dbms. In *Proceedings of ICDE*, pages 466–475, 2007.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[5] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 54–65, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[7] boost.org. Boost c++ libraries. `http://www.boost.org/`, 2010.

[8] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[9] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, pages 268–279, New York, NY, USA, 1985. ACM.

[10] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):1–39, 2009.

[11] A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. *Proc. VLDB Endow.*, 1:502–513, 2008.

[12] S. N. Khoshafian, G. P. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A Query Processing Strategy for the Decomposed Storage Model. In *Proceedings of the Third International Conference on Data Engineering*, pages 636 – 643, Washington, DC, USA, 1987. IEEE Computer Society.

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

[14] M. Zukowski, P. A. Boncz, N. Nes, and S. Heman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, 2005.

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

[16] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[17] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[18] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.