On Improving Data Skew Resilience In Main-memory Hash Joins

Puya Memarzia
Univeristy of New Brunswick
Fredericton, Canada
pmemarzi@unb.ca

Suprio Ray
Univeristy of New Brunswick
Fredericton, Canada
sray@unb.ca

Virendra C Bhavsar
Univeristy of New Brunswick
Fredericton, Canada
bhavsar@unb.ca

ABSTRACT

Main memory hash joins are an important category of in-memory joins. However, the performance of these joins can be hindered by dataset skew, shuffling, and load balancing. We conducted a comprehensive study on the effects of dataset skew on four hash join algorithms. We show that hash joins are acutely affected by dataset skew, and the performance gets worse with shuffled data. To address these issues, we propose non-partitioning hash joins using two different hash tables. First, we use a separate chaining hash table that is based on an existing implementation that we have modified. This version outperforms the original implementation on skewed datasets by up to three orders of magnitude. Second, we propose a novel hash table for hash joins, called Maple hash table. We demonstrate that this hash table is better suited to skewed and/or shuffled datasets. Moreover, this approach further improves performance by up to 17.3×.

CCS CONCEPTS

• **Information systems** → *Data structures*; *Database query processing*; *Join algorithms*;

KEYWORDS

hash tables, in-memory joins, dataset skew, intra-query parallelism

ACM Reference format:
Puya Memarzia, Suprio Ray, and Virendra C Bhavsar. 2018. On Improving Data Skew Resilience In Main-memory Hash Joins. In *Proceedings of 22nd International Database Engineering & Applications Symposium, Villa San Giovanni, Italy, June 18–20, 2018 (IDEAS 2018),* 10 pages.
https://doi.org/10.1145/3216122.3216156

1 INTRODUCTION

Databases are one of the key technologies that power the information age. In recent years, the need to store, retrieve, and process data efficiently, has become vital in all spheres of human activities. Furthermore, the variety of applications and problems that rely on database performance has grown dramatically. The importance of database efficiency and performance is greater than ever before. Thus, database performance continues to be a hot research topic. Joins are an important operation in many database systems. Hash joins utilize hash tables to speed up the process. Main memory database systems have long used hash joins to improve performance [14]. Recent advances in data systems and computer hardware have spurred further research in this area. As memory becomes cheaper and denser, main-memory hash joins are increasingly prescribed to speed up existing systems.

One of the key challenges facing database developers is how to consistently achieve good performance. In their in-depth analysis of in-memory hash join algorithms, Blanas et al. [8] acknowledged the importance of data skew. They showed that a simple no partitioning hash join algorithm can outperform other join algorithms. However, they only considered a basic *1:N* join case that involves matching a primary key (unique and ordered) with a foreign key from another table. Joins can involve non-key columns that do not enforce uniqueness or any particular ordering.

Popular relational database benchmarks such as TPC-H [9] generally focus on querying data that is uniformly distributed and non-skewed. However, it has been shown that these cases are not necessarily representative of real-world applications [10, 18]. For example, the size of cities and the length and frequency of words can be modeled with Zipfian distributions, and measurement errors often follow Gaussian distributions [15]. Furthermore, skewed build keys can be encountered as a result of parallel multi-way joins [7], joins on non-primary key columns, and complex queries. It is quite common to observe dataset skew in the output of a join operation, and this result-set may need to be joined with several other intermediate results or tables. The performance impact of dataset skew is frequently overlooked in favor of heavily optimizing other aspects, such as the effects of cache and TLBs [20], NUMA characteristics [23], architecture awareness [5, 8], memory-efficiency [6], and transactional memory [28].

To demonstrate the acuteness of the data skew problem, we conduct an experiment comparing the hash join performance of a non-skewed dataset that is similar to that used in [8] and a skewed dataset that we generated (build table skew is the only variable). The experiment uses the hash join configurations and code provided by [8], which we describe in Section 2.2. We run the experiment on a modern processor based on the Skylake architecture (for specifications, see section 6.1). In Figure 1(a) the results show significant performance hits on all hash join configurations by nearly four orders of magnitude, when the dataset is skewed. As we show in section 4, this issue is caused by cache misses incurred from traversing long pointer-linked chains of key-value pairs. Thus we illustrate how dataset skew can severely hinder hash join performance if it is not mitigated.

To further study the effect of data skew on hash joins, we propose a set of sixteen datasets that vary in terms of distribution, skew, and shuffling. The details of these datasets are presented in Table 3.

IDEAS 2018, June 18–20, 2018, Villa San Giovanni, Italy

ACM ISBN 978-1-4503-6527-7/18/06…$15.00
https://doi.org/10.1145/3216122.3216156

To our knowledge, no prior work has used such a comprehensive series of datasets to evaluate in-memory hash joins. We explore different variants of dataset skew, vary the correlation between the build and probe tables, and examine how the order of the keys can affect the join algorithm.

Query optimizers generally rely on various statistics about the tables to reliably predict the cost of different parts of a query. Choosing the right tool for the job can be challenging. In typical real-world applications, we cannot control how and where our data will be skewed. To address the impact of dataset skew on hash joins, we focus our attention on the design of the hash table.

Separate chaining hash tables remain one of the most popular choices due to their ease of use and flexibility. They have been used in many existing hash join implementations such as [4, 5, 8, 23, 28]. We demonstrate that the hash tables used in their implementations are not ideal for joins on skewed datasets. In order to improve performance, we modify the existing hash table (based on separate chaining) from [8]. We show how this modification results in shorter chain length, simpler result materialization, and significantly improves total join time. We demonstrate an example of this in Figure 1(b) The modified hash table improves performance by storing the values associated with each key in a contiguous in-memory vector. This significantly reduces the number of lookups required to find a key because there are fewer elements in each bucket chain (this is covered in detail in Section 4). The hash join configurations are covered in more detail in Section 2.2.

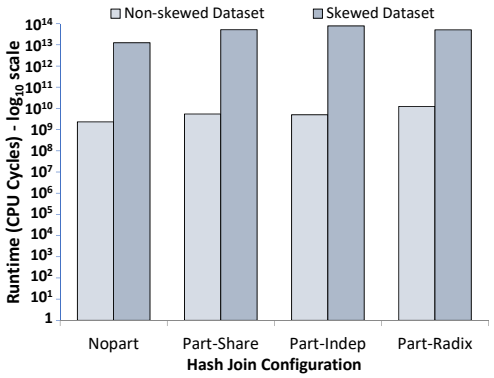
To further improve hash join performance, we propose Maple hash table, a novel concurrent hash table based on cuckoo hashing. Maple hash table uses a unique hashing technique to determine a key's destination table, bucket, and index within the bucket. This multi-stage hashing approach reduces contention and collisions, without increasing the number of required lookups. We show that Maple Hash table outperforms a state-of-the-art concurrent hash table implementation from Intel (see Section 5).

We demonstrate that non-partitioning hash joins based on Maple hash table can significantly improve performance, particularly when the data is not ordered. Join time is faster by up to 17.3× in the best case, and slower by less than 0.2× in the worst case, compared to partitioning hash joins using our improved separate chaining hash table. This presents an opportunity for a query planner to choose a better hash join method and hash table, based on the data characteristics.

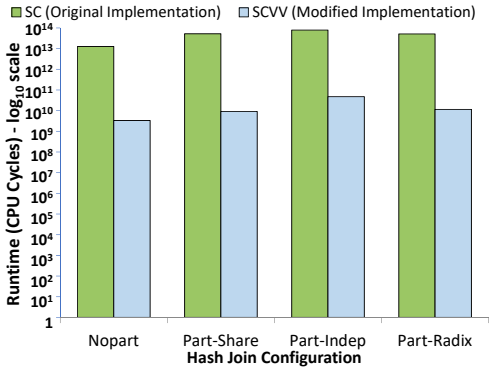
To show the effectiveness of our approach on different processor architectures, we conduct experimental evaluations on three distinct hardware platforms. The hardware details are presented in Table 1. Our approaches achieve consistent performance gains, without the need to manually tune the parameters for each hardware architecture.

The key contributions of this paper are:

- We show how dataset skew can severely hinder hash join performance if it is not mitigated.
- We design a series of datasets that extends the variety of data distributions and relationships that are evaluated.
- We implement a modified version of the hash table used by Blanas et al. [8] that significantly improves join performance with skewed datasets.



(a) Highlighting the performance impact of dataset skew and shuffling - Comparing a non-skewed ordered dataset and a Gaussian skewed dataset



(b) Comparing separate chaining hash table (SC) from [8] against our modified version (SCVV) - our approach is over three orders of magnitude faster on the skewed dataset (details in Section 4)

Figure 1: Experiments using hash join configurations from [8] - Skylake - 8 threads

- We present joins using Maple hash table, a novel hashing technique based on cuckoo hashing that further improves performance on shuffled workloads.

The remainder of this paper is organized as follows. In Section 2 we discuss the related work. In Section 3 we describe the issues that arise from skewed datasets. We present two solutions to solving this problem in Sections 4 and 5. The experimental setup and results are presented in Section 6. We conclude the paper in Section 7.

2 RELATED WORK

Research on main-memory hash joins has flourished in recent years. Numerous publications have explored different algorithms, workloads, and architectures. In this section, we summarize recent works on hash joins and hash tables.

2.1 Hash Joins

A key inspiration for our work is the in-depth analysis on main-memory hash joins presented by Blanas et al. [8]. The authors implement a family of hash join algorithms (summarized in Section 2.2), which we adopt for our work. Their experiments evaluate datasets with skew on the probe relation. The results indicate that

a simple non-partitioned join algorithm using a separate chaining hash table frequently outperforms other approaches, particularly when probe skew is introduced.

Using the framework implemented by [8] as a base, Balkesen et al. [5] make the case for fine-tuning radix hash join to the hardware. The authors extensively analyze the performance of Radix and non-partitioning in-memory hash joins, using workloads adapted from [20] and [8]. Their results provide valuable insight on the role CPU architecture plays in hash join performance. Our work also builds on the framework from [8], but instead focuses on addressing dataset skew on *hardware-oblivious* hash joins. Interestingly, the authors predict that hardware advancements will eventually eliminate the need for fine-tuning in the future.

In [6] Barber et al. present two new hash tables for in-memory hash joins. The authors focus on improving hash join memory efficiency. They note that their approach cannot handle M:N joins.

Shanbhag et al. [28] describe a hash join implementation that uses hardware transactional memory and takes advantage of spatial locality in the data. They also note the importance of evaluating both ordered and shuffled datasets.

2.2 Hash Join Configurations

We employ the same hash join algorithms proposed by Blanas et al. in [8]. These consist of one non-partitioning hash join and three partitioning hash join variants. The following is a short description for each hash join variant along with the shortened names used in our charts.

(1) **No partitioning join (Nopart)**: all threads create a shared hash table from the build relation. This hash table is then probed concurrently.

(2) **Shared partitioning join (Part-Share)**: both relations are divided into partitions shared by all threads. Locks are used to facilitate concurrent access.

(3) **Independent partitioning join (Part-Indep)**: all threads participate in partitioning both relations, but the partitions are private and cannot be accessed by other threads. Consequently, locks are not needed.

(4) **Radix partitioning join (Part-Radix)**: parallel Radix Join with dynamic load balancing, as described by Kim et al. [20]. Each input relation is split into a hierarchy of partitions using their least significant bits (LSB). The resulting partitions are then joined using a hash table.

2.3 Hash Tables

There are many variants of hash tables, but not all hash tables are suitable for hash joins. Cuckoo hashing was first proposed by Pagh et al. [26]. It guarantees constant lookup, update, and deletion times, but insertion times are amortized. Kirsch et al. [21] use a stash to store elements that could not be inserted normally. In [22] Kumar et al. employ a hierarchy of hash tables to provide deterministic insertions for cuckoo hashing. In [24], researchers from Intel labs present a concurrent cuckoo hashing technique which utilizes hardware transactional memory (HTM). The authors leverage this hardware feature to implement atomic modification of shared data structures. In their experiments they show that this approach outperforms Google's dense_hash, Intel TBB, and C++ unordered_map.

These hash tables are not designed with hash joins in mind, because the "insert" operation typically overwrites existing values. In a hash join, this may result in incomplete results.

3 THE IMPACT OF DATA SKEW

To support hash joins, the hash table implementation must be modified to support duplicate keys. In related work by Blanas et al. [8], the authors used a hash table implementation based on separate chaining, and presented an in-depth analysis on the impact of data skew on hash join performance. However, this analysis was limited to data skew on the probe table. The authors concluded that partitioning hash joins are less resilient to dataset skew. In order to examine this problem further, we developed an extensive set of datasets, and evaluated these datasets with the code provided by [8]. We discovered that joins with highly skewed build relations were significantly more time consuming than joins with similar probe skew.

Separate chaining hash tables are used for a wide variety of applications due to their simplicity and flexibility. The main concept behind separate chaining is to store items in an array of buckets, using a hash function (often a fast and simple solution such as modulo) to determine the bucket number, and resolve collisions by chaining items together. Separate chaining is often used for hash join applications because insert operations are very fast (we do not need to traverse the whole chain to add a new item), and the insert operations do not fail as long as there is available memory. If the input data is not skewed (as is the case in [8]), non-partitioned hash joins with separate chaining generally perform well. However, duplicate keys in the build table results in long chains of items in the hash table, regardless of the hash function used. This can significantly hinder performance. We first observed this behavior when testing the code from [8] with skewed datasets. Initially, we solve this problem by modifying the hash table data structure and data operations.

4 SEPARATE CHAINING WITH VALUE-VECTORS (SCVV)

As we discussed in the previous section, dataset skew can pose a significant obstacle for hash join performance. We initially address this issue by modifying the separate chaining hash table from [8] to store/read multiple values per key in contiguous *value vectors*. This simple solution is very effective against dataset skew and can also be extended to other data structures.

Figure 2 depicts an example comparing our modified separate chaining hash table to the original version. Consider a chained hash table with 10 buckets and a modulo hash function. During the hash join probe phase, we query key 7. We have to traverse all five elements in the chain to look for matching keys. In the modified version with value vectors, we find a match on the first lookup, and we find all other values by traversing the value vector. In this example, finding all the matches for key 7 will take 2 reads, versus 5 reads in the original version. This issue does not depend on the hash function. As we show in the experiments, the overall impact on join time can be very severe.

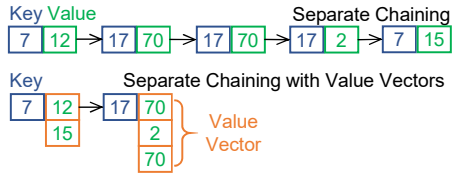


Figure 2: Example: Comparison of a bucket chain with and without value vectors (modulo hash function).

4.1 Lookup and materialization costs

We do an analysis on separate chaining lookup costs to provide some intuition on how SCVV can improve performance. Let's assume K keys have been inserted into the hash table. The worst case scenario occurs when all the keys end up in the same bucket. In this situation, we have to perform K lookups for K keys in the build table for every key that exists in the probe table.

In order to simplify this analysis, we focus on what happens while probing the keys in a bucket chain. In the case of regular separate chaining, we have to check every single key in the bucket, because we cannot guarantee that another matching key does not exist in the bucket.

With a separate chaining hash table (SC), if a bucket has K keys, the number of key lookups needed until we find a match is as follows: $C_{SC} = K$.

In a modified separate chaining hash table with *value vectors* (SCVV), we only need to traverse the chain until we find a match. For a bucket with K chained **unique** keys, the average number of lookups is half the number of lookups required by basic separate chaining: $C_{SCVV\_unique} = \frac{K}{2}$.

However, when the keys contain duplicates due to dataset skew, the maximum chain length is determined by the number of unique keys that hash to a bucket. If we assume that D is the average degree of key duplication, the lookup cost is calculated as $\frac{K}{D*2}$ and by factoring in the average cost of reading the value vector once we have found a match, we get the cost of materialization: $C_{SCVV\_skewed} = \frac{K}{D*2} + D$.

In conclusion, value vectors reduce the number of key lookups needed, and once we find a matching key, we can stop traversing the chain, and output all the resulting tuples. Henceforth, we use SCVV as our new baseline.

4.2 Evaluation and Overhead

We compare the original implementation by Blanas et al. [8] with our modified version using value vectors. In order to provide an "apples-to-apples" comparison, we only modify the hash table used, and leave the rest of the code unchanged for this experiment. A performance comparison of hash joins using SCVV and the original hash table implementation (SC) with **skewed** datasets is shown in Figure 1(b). Our approach is several orders of magnitude faster than the original code when the dataset is skewed (up to 5726× faster). As discussed in Section 4.1, SCVV produces shorter chains compared to SC. This significantly lowers the number of lookups for both the build and probe phases.

We have demonstrated that SCVV is faster than SC on skewed datasets, but what about datasets that are not skewed at all? The main advantage of using a value vector is to store and retrieve multiple values per key. When there are no repeating keys in the dataset, that advantage is lost, and the size of the hash function modulus becomes an important factor. Furthermore, despite the specialized nature of the value vector, it bears additional overhead compared to a much simpler value variable. SCVV incurs a performance penalty of 16% when the build table keys are unique and fully ordered (a sequence of the numbers 1 to N). As a result, we would still choose SC in such rare cases. We have designed a new baseline implementation that can handle skewed datasets, and our next goal is to tackle shuffled (non-ordered) datasets.

5 MAPLE HASH TABLE (MH)

As part of our research on hash tables for hash joins, we explored the possibility of using cuckoo hashing for this purpose. As we mentioned in Section 2, cuckoo hashing was originally proposed by Pagh et al. [26]. Its core concept is to store items in one of two tables, each with a corresponding hash function (this can be extended to N tables). If a bucket is occupied by another item, the existing item is displaced and reinserted into the other table. This process continues until all items stabilize, or an arbitrary threshold is crossed. Cuckoo hashing provides a guarantee that reads take no more than two lookups. Its main drawback is relatively slower and less predictable insert operations, a lack of concurrency, and the possibility of failed insertions. Consequently, several variants of cuckoo hashing have been proposed to resolve these drawbacks. A concurrent hash table based on cuckoo hashing was proposed by Intel in [24] (henceforth called "Intel Libcuckoo"). It supports high-performance concurrent operations with multiple readers and writers. By leveraging hardware transactional memory (HTM) introduced in Intel's new Haswell chipset, as well as several carefully engineered optimizations, it achieves the best insert performance amongst all hash tables. However, the availability of special hardware features such as HTM cannot be taken for granted. Furthermore, the performance of Intel Libcuckoo suffers significantly when the distribution of the dataset is skewed. In their experimental evaluation [24] Intel Libcuckoo used a uniform distribution of keys. As we know, real world datasets may be non-uniform or skewed [17].

We introduce Maple hash table, a novel extension to the bucketized cuckoo hash table approach that is designed for concurrency, and insert efficiency. In our approach, we use a fast multi-stage hashing technique to determine an incoming item's location, effective load-balancing to decrease collisions and lock contention, a semi-optimistic locking strategy that acquires fewer locks without incurring race conditions, and optimizations to improve concurrency and cache usage.

In the next section (Section 5.1) we describe the Maple hash table data structure. We then elaborate on the hash functions in Section 5.2. Finally, we describe how the hash table functions and concurrency control are implemented in Section 5.3.

5.1 Data Structures

Figure 3 depicts a general overview of our cuckoo hash table. The hash table uses 4-way set-associative buckets, which are implemented as contiguous arrays of key-value pairs. Other configurations are possible but 4-way provides a nice balance of performance and memory usage, while also ensuring cache alignment. As we show in Subsection 5.2, an element can be found without reading through the whole bucket.

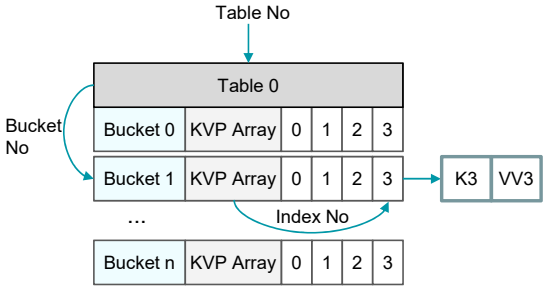


Figure 3: Maple Hash Table Data Structure - a key is found without the need to probe all the elements in the bucket

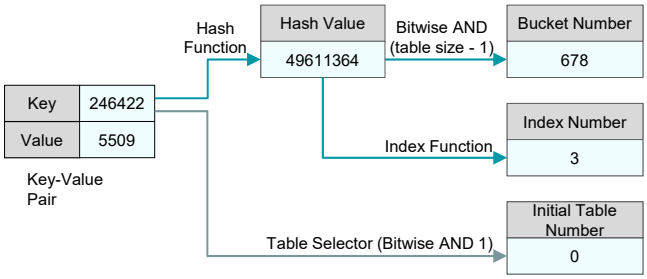


Figure 4: Example of finding a location for a key using multi-stage hashing to determine table, bucket, and array index

The combination of the table, bucket, and index numbers gives us the precise location of a key-value pair. For each key, there are only two possible valid locations within the table. That means that unlike other methods that combine cuckoo hashing with linear probing, such as [24], we can retrieve a key-value pair by looking up a maximum of two slots as opposed to $2N$ (where N is the number of slots per bucket). This feature further reduces the cost of lookups in buckets that are larger than the cacheline, as parts of the bucket that cannot contain the key are skipped. When the table is first created, all keys and values are initialized to zero. We reserve zero to indicate that a key-value pair is empty or deleted.

5.2 Hash Functions

A good hash function strikes a balance between computational complexity and its effectiveness at reducing collisions. As noted in [27] and [1], Murmur hashing is widely used due to its speed and acceptable hash value quality in most situations. Our own experiments with a wide variety of hash functions confirm that Murmur is a suitable choice. However, in some test cases (skewed datasets) its performance was not satisfactory. Due to this, we implemented a multi-stage hashing approach that is effective and computationally cheap. Figure 4 depicts an example of how each index is computed.

An incoming key is passed through the hash function to produce the hash value. To determine the bucket number, we calculate the modulo of the hash value to the table size. Since the table size is a power of two, we calculate this much faster by using the well-known technique of replacing the modulo with a bitwise AND operation. A function called the *indexgen* is used to determine the index within that bucket. In practice, this approach works well when the *indexgen* function can be computed much faster than the hash function.

Algorithm 1: Maple hash table insert algorithm

```
   Data: newKVP is the key-value pair being inserted
   Result: Return true if successful, otherwise false (rehash)
 1 if find(KVP) then append(KVP);
 2 else
 3     stepcounter ← 0 ;
 4     maxsteps ← log(tablesize);
 5     tableno ← newKVP.key bitwise & 1;
 6     while stepcounter < maxsteps do
 7         switch tableno do
 8             case 0 do
 9                 hashval ← hash1(newKVP.key);
10                 bucketno ← hashval & (tablesize - 1); index ←
                    indexgen(hashval);
11                 lock lockarray[bucketno];
12                 if table0[bucketno][index] is empty then insert
                    newKVP in table0;
13                 unlock lockarray[bucketno];
14                 return true;
15                 else
16                     if key exists then append(newKVP) and
                        return true;
17                     evict existing KVP as oldKVP and insert
                        newKVP;
18                     unlock lockarray[bucketno];
19                     Set newKVP←oldKVP;
20                     tableno ← 1;
21             case 1 do
22                 (Repeat case 0 steps for table 1)
23         increment stepcounter by 1;
24 return false;
```

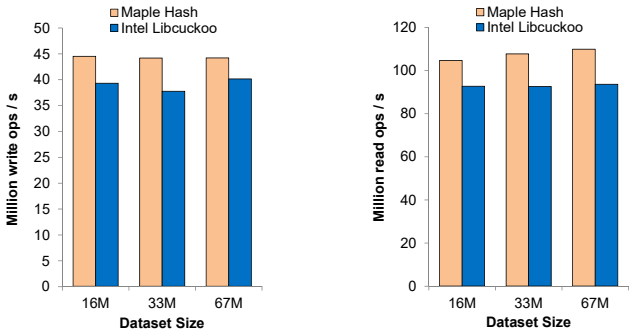
5.3 Concurrent Implementation

Cuckoo hashing is relatively easy to implement in serial applications. However, the original design by Pagh et al. [26] did not propose an efficient parallel design. Instead, the authors propose calculating two hash values in parallel. This approach does not scale well on modern processors, which generally have over 4 cores.

Our implementation leverages data parallelism to scale up on multicore processors. As noted in [16], there is no stable hierarchy between locks, and variables such as workload, available resources, and hardware specifications, need to be taken into account. For concurrency control, we opt to use light-weight spinlocks on a per-bucket granularity.

To ensure consistency, all threads must finish building the hash table before it can be probed for matching keys. This allows us to employ a locking strategy that is less stringent, as concurrent read and write operations are not required. Our semi-optimistic locking strategy reduces lock acquisition and contention.

A listing of our *insert* algorithm is depicted in Algorithm 1. If a key already exists in the hash table, the value of the incoming key-value pair is appended instead. Our preliminary experiments



(a) Insert throughput (b) Read throughput

Figure 5: Comparing Maple hash table and Intel Libcuckoo - uniform dataset - Skylake - 8 threads

revealed that having all threads begin inserts from one table could result in uneven load distribution and higher lock contention. The table selector selects the initial table to insert a new item, based on the key's least significant bit. We use the least significant bit because it is fast and effective. In a worst case scenario (all keys are odd or even), the algorithm behaves like most cuckoo hashing approaches that start insertions from the same table. It is possible to further randomize this approach, but we found it to be adequate.

Rather than locking every bucket along the cuckoo path, we lock the bucket that will be modified. To ensure consistency, the algorithm checks for race conditions before modifying an item. Successfully inserting a new item into the hash table will store that item in only one of two possible locations. This ensures good read performance when probing the table for matches.

5.4 Performance Evaluation

We compare the performance of Maple hash table against the state-of-the-art concurrent hash table, Intel Libcuckoo. The datasets consist of 16M, 33M and 67M random records generated from a uniform distribution. Figures 5(a) and 5(b) depict multi-threaded insert and read throughput respectively. Our results show that Maple hash table is modestly faster than Intel Libcuckoo by up to 17%.

6 EVALUATION

We now evaluate the effectiveness of the hash join approaches with our hash tables. Our experiments build on the same code base as [8], which was also adopted by [4]. Our contributions include implementing and integrating the SCVV and MH hash tables into the benchmark and investigating their performance on a broad range of datasets.

It was shown in previous work that hardware architecture can play an intricate role in hash join performance [5]. Inspired by this work, we evaluate our experiments on three different hardware architectures. For convenience, we summarize the hash join configurations in Table 2.

We present the platform specifications of each machine in the next section (Section 6.1), and describe the characteristics of our synthetic datasets in Section 6.2.

6.1 Platform Specifications

We evaluate the experiments on three different processor architectures: Intel Skylake, Intel Harpertown (based on the Penryn architecture), and AMD K8. Our intention is to ensure that the results are reproducible and not tuned for a particular set of hardware. The Harpertown and AMD machines run Ubuntu 14.04 LTS, and the Skylake machine uses Ubuntu 16.04 LTS. To ensure consistency in code compilation, we configure all machines to use version 6.3.0 of the G++ compiler, and enable the -O3 optimization flag.

Table 1: Experimental setup specifications

CPU	Cache	RAM	TLB (4KB pages)
Intel Core i7 6700HQ (Skylake)	1MB L2 6MB L3	16GB DDR4	L1 DTLB: 64 entires L2 STLB: 1536 entires
Intel Xeon E5472 (Harpertown)	12MB L2 No L3	16GB DDR2	L1 DTLB: 16 entries L2 DTLB: 256 entries
AMD Opteron 8220 (K8)	2MB L2 1MB L3	128GB DDR2	L1 TLB: 32 entries L2 TLB: 512 entries

Table 2: Hash Join Configuration Key

Short Form	Hash Join Variant	Hash Table
Nopart_SCVV	No partitioning	SCVV
Nopart_MH	No partitioning	Maple
Part-Share_SCVV	Shared partitions	SCVV
Part-Indep_SCVV	Independent partitions	SCVV
Part-Radix_SCVV	Radix partitioning join	SCVV

The first machine contains an Intel Skylake quad core processor with hyper-threading, 16GB of RAM, and a 500GB SSD. The CPU contains 256KB of L2 cache per core, and 6MB of L3 cache that is shared by all cores. The next machine uses an Intel Harpertown (based on the Penryn architecture) quad core, and 16GB of DDR2 RAM. Each pair of cores has access to 6MB of L2 cache. Lastly, we test an AMD machine with eight K8 CPUs combined with 128GB of RAM. The hardware specifications are summarized in Table 1.

6.2 Datasets

In [8], the authors use dataset cardinalities of 16M and 256M tuples to mimic common decision support settings (1:16). Subsequent works by other authors have adopted this workload [4, 5, 28]. We adopt this cardinality as the baseline for our datasets. We extensively expand the range of datasets to include distributions not covered by prior work, noting that even popular benchmarks such as TPC-H are not representative of all real-world workloads [10, 18].

In most databases, the primary key is automatically generated in ascending order. However, there is no guarantee that the join keys will be ordered and/or uniformly distributed. Although some recent works examine dataset skew on the probe relation, to our knowledge none of the literature explore such an extensive set of datasets on main-memory hash joins. Another aspect to consider is the way the keys are ordered. Prior work by Shanbhag et al. [28] highlights the importance of studying dataset shuffling.

Table 3 provides details on how the datasets are designed. All build tables contain 16M records, and the probe table size depends on the correlation. For each dataset, we also generate a *shuffled* version that is reordered using a uniform random function. The datasets are otherwise denoted as *ordered*. It is worth noting that if the build keys are drawn from a skewed distribution (Zipf or Gaussian), they will not be in sorted order. However, on ordered datasets the correlating keys in the probe table will follow the same order as the build table, resulting in better data locality.

Table 3: Dataset details

Dataset Name	Build Key Generation	Build to Probe Key Correlation	Parameters
Sequential 1:N	sequential unique keys	keys have exactly N matches in probe table	N=16, key range 1 to 16M
Random near 1:N	randomly repeating sequence	keys have exactly N matches in probe table	N=16, uniform random between 1 and 5
Gaussian 1:N	sequential unique keys	relationship follows a Gaussian distribution	mean=0.015, stdev=0.3, multi=10
Gaussian M:N	Gaussian skewed keys	relationship follows a Gaussian distribution	mean=0.015, stdev=0.6, multi=10000
Gaussian near M:K	Gaussian skewed keys	Random K matches where $1 < K < N$	N=16, mean=0.015, stdev=0.6, multi=10000
Zipf 1:N	sequential unique keys	relationship follows a Zipf distribution	Zipf skew=2.0, size=16M
Zipf M:N	Zipf skewed keys	relationship follows a Zipf distribution	Zipf skew=2.0, size=16M
Zipf near M:K	Zipf skewed keys	Random K matches where $1 < K < N$	N=16, Zipf skew=2.0, size=16M

6.3 Results and Discussion

We conduct comprehensive experiments to analyze how dataset skew and data shuffling affect hash join performance. These datasets are designed to focus on the effects that we are interested in studying. To this end, we process each of the datasets with the hash join configurations mentioned in Section 2.2. The join operation concludes when the resulting tuples have been written to memory. As our main focus is in-memory performance, we omit the final step of writing the result to the disk. This is consistent in all the experiments to ensure "apples-to-apples" comparisons.

We present our performance evaluations in Subsections 6.4 to 6.7. In each subsection, we focus on one parameter (such as build skew), and keep all other parameters constant. Throughout the results we refer to non-partitioning join using the Maple hash table as *MH*, and Separate Chaining with Value Vectors as *SCVV*. Finally, we examine how CPU architecture can affect the results in Section 6.7. We measure the CPU cycles for each hash join phase, using the timers from [8]. These timers measure CPU cycles, allowing results from different datasets can be compared.

6.4 Build Table Skew

We start off by investigating how build skew affects performance. We evaluate four different variations of build table skew, while maintaining the 1:16 probe cardinality used in prior work [8].

In Figure 6 we present the average runtimes for each ordered dataset, and in Figure 7 we present the results for the shuffled versions of these datasets. The charts are arranged in order of increasing build skew, from left to right. We now examine the behavior of each dataset in this category.

6.4.1 Sequential 1:N dataset. A basic scenario is to consider a build table with unique sequential keys. We note that no partitioning with SCVV is the fastest configuration when the dataset is ordered, and that MH outperforms all other configurations when the dataset is shuffled. This is a trend that we frequently observe throughout our experiments. We take a closer look at the effect of shuffling in Section 6.6. Overall, the non-partitioning hash join configurations provide the fastest runtimes.

6.4.2 Random near 1:N dataset. This dataset contains mildly skewed build keys. This is achieved by randomly repeating sequential keys between 1 and N times until 16M records are created. This is the only example in all our experiments where Radix hash join outperforms all other configurations. When the keys are ordered, MH and SCVV offer very similar performance and are the fastest configurations, and Part-Radix is the worst configuration. When the keys are shuffled, all configurations take a performance hit, but Part-Radix ends up as the fastest configuration followed by MH. Previous studies have noted that Radix join is prone to load imbalance [8], and shuffling may partially alleviate this.

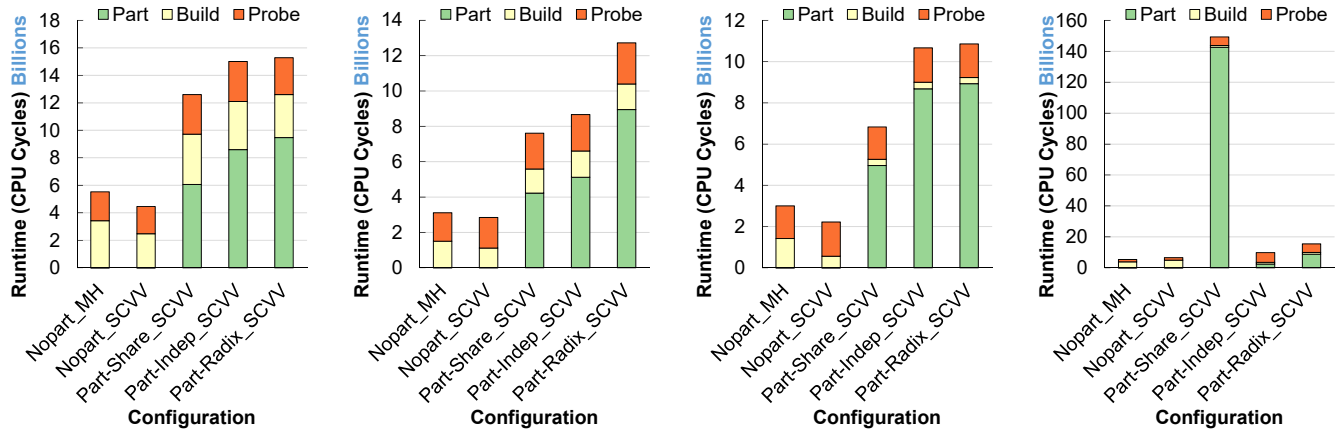
6.4.3 Gaussian M:N dataset. For this dataset, the build keys approximate a half-normal Gaussian distribution with a standard deviation of 6000. The keys cover a broader range of numbers and frequencies, compared to the Zipf skewed keys. The data locality in the probe table is lost when the keys are shuffled. The results show that SCVV provides the fastest runtimes, followed by MH. The ordered results follow the same trend as the sequential *1:N* dataset. When the dataset is shuffled, all configurations take longer to complete the join. The Part-Indep configuration suffers the greatest slowdown during the partitioning phase.

To understand why this happens, consider that Part-Indep creates per-thread partitions. This eliminates the overhead of synchronization, but also results in a much larger total number of partitions that may no longer fit in the cache and TLB. These private partitions are then merged after the partition phase. These characteristics delay the availability of data. The shuffled dataset further increases the number of partitions created per thread, which magnifies the problem.

6.4.4 Zipf M:N dataset. In order to study a case of extreme data skew on the build table, we use a Zipf distribution with a high skew parameter of s= 2.00. The runtime results show the Part-Share configuration performing worst by a considerable margin. If we look at the join breakdown by phase, we see that the partition phase is the main bottleneck. The Part-Share configuration creates partitions that are shared among all threads and protects concurrent inserts using a latch. This particular Zipf distribution contains keys that cover a relatively narrow range of values. Consequently, most keys belong to the same partition. This results in high lock contention and serialization of the partitioning phase, as all threads will try to insert into the same partition. The same trend is observed after shuffling the data because the keys will be inserted into the same shared partition regardless of how they are distributed. Other configurations such as MH and SCVV offer finer lock granularity or avoid locks altogether in the case of Part-Indep. We can conclude from this that the Part-Share configuration is unsuitable when the majority of build keys occupy a narrow range of numbers.

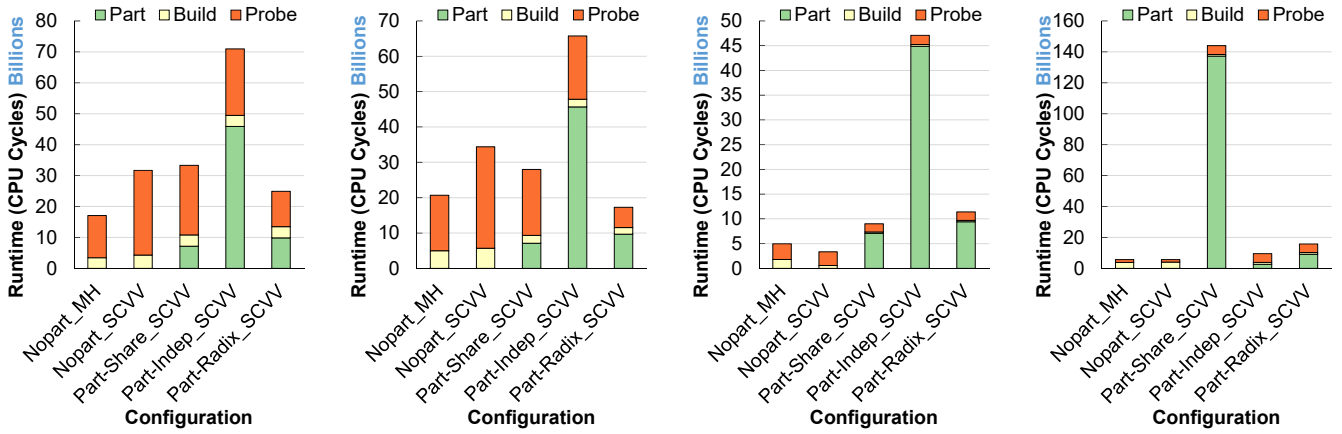
6.5 Probe Table Skew

We now examine probe table skew. In these experiments, we keep the build skew constant, and vary the probe skew. We examine a total of four different probe skew distributions, that are divided into two categories in order to keep the build skew constant when comparing the results.



(a) **Sequential** - 1:N (b) **Random** - near 1:N (c) **Gaussian** - M:N (d) **Zipf** - M:N

Figure 6: Hash Join run time with Variable Build Skew on *Ordered* Datasets - Skylake - 8 threads



(a) **Sequential** - 1:N (b) **Random** - near 1:N (c) **Gaussian** - M:N (d) **Zipf** - M:N

Figure 7: Hash Join run time with Variable Build Skew on *Shuffled* Datasets - Skylake - 8 threads

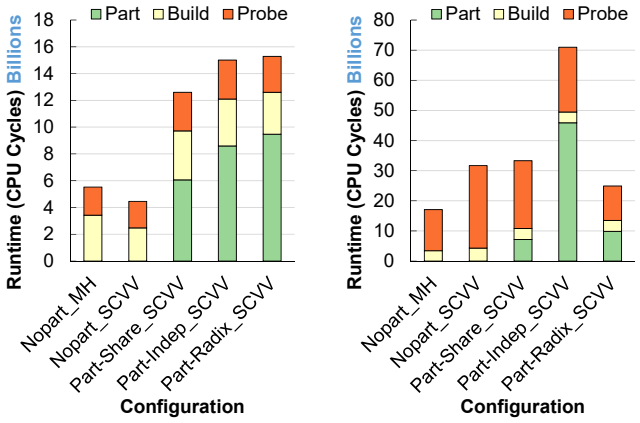
We generate the build table using non-skewed keys, and vary the correlation with the probe table to either be exactly *1:N*, Zipf skewed, or Gaussian skewed. The results are shown in Figure 8, with each dataset paired next to its shuffled variant. The results continue earlier trends of showing SCVV leading in ordered datasets, and MH provided the fastest times for shuffled datasets. The sequential *1:N* dataset produces exactly N=16 keys in the probe table for every key in the build table. The Zipf and Gaussian *1:N* datasets pick the number of matches out of their respective skewed distributions, and are capped at 16 matches. As a result, they generate fewer total probe keys. Prior works showed that non-partitioning hash joins benefit from skew on the probe table compared to non-partitioning hash join variants [5, 8]. We observe this in our results, but only when the datasets are ordered.

We also explore datasets that are generated with a near *M:K* relationship. In these datasets, the number of correlating matches for each build key is K, which is a number randomly chosen between 1 and N. The random K number is chosen using a uniformly random distribution. Using a near *M:K* distribution results in a smaller probe table, and thus relatively faster join speed over *M:N*. We omit the charts for this category, as there are no noteworthy changes to the relative performance of the hash join configurations.

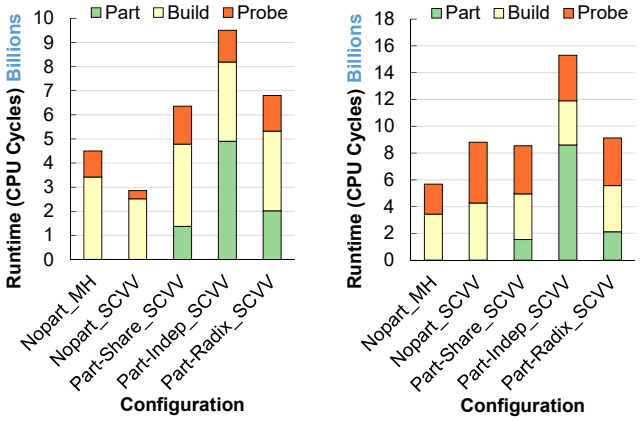
6.6 Effect of Dataset Shuffling on Hash Tables

Every dataset has an ordered version and a shuffled version which is shuffled using a uniformly random distribution. It is observed throughout the results that all configurations take a performance hit going from an ordered dataset to a shuffled dataset. This trend applies even when the build keys are randomly drawn from a skewed distribution, and repeated in the probe table N times. This occurs because repeated keys in the probe table would no longer be probed in the order that they were generated. As a result, we are significantly less likely to find the data in the cache or TLB.

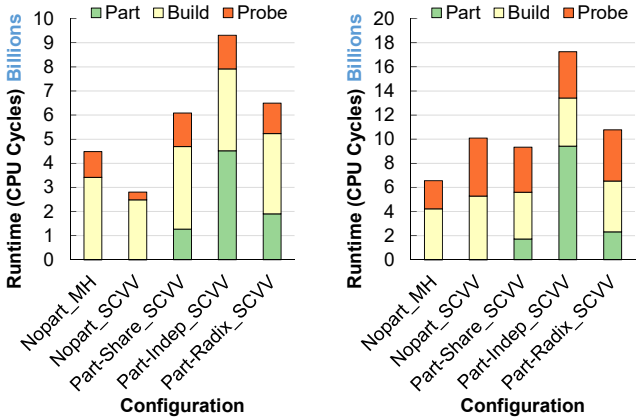
There is also a clear trend of MH performing better when the data is shuffled, and SCVV taking the lead when it is ordered. This trend is of particular interest to us, as it presents an opportunity



(a) **Sequential** - Ordered (b) **Sequential** - Shuffled



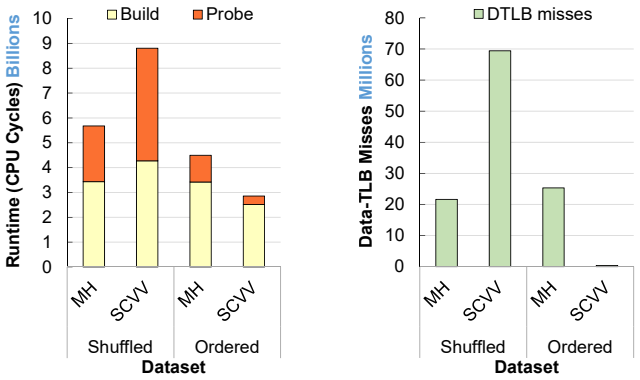
(c) **Zipf** - Ordered (d) **Zipf** - Shuffled



(e) **Gaussian** - Ordered (f) **Gaussian** - Shuffled

Figure 8: Hash Join Runtime with Variable Probe Skew on 1:N Datasets - Skylake - 8 threads

for a query planner to choose a hash table based on whether or not the data is shuffled. In order to gain better insight this, we measure the TLB and cache misses. In Figure 9(a) we compare runtimes and in Figure 9(b), we show the corresponding d-TLB misses. When



(a) Runtime (b) Data-TLB Misses

Figure 9: Performance impact of shuffling on MH and SCVV - Zipf 1:N dataset - Skylake - 8 threads

the dataset is ordered, the SCVV data structure is also accessed in a sequential manner. As a result entries that are evicted from the dTLB will not be accessed again during the course of the current phase. Due to the fact that the size of the dataset far exceeds the number of entries that the TLB can hold (64 4KB pages), the chance of incurring a TLB miss for every lookup is greatly increased on a shuffled dataset.

SCVV greatly benefits from ordered data, as it uses a modulo hash function, and the buckets are accessed in the same order as the keys. This benefits lookup times due to increased memory locality and fewer cache misses. When the keys are shuffled, the threads access the hash table is a random and unpredictable fashion. As a result, SCVV suffers from increased TLB and cache misses, and potentially more lock contention. Shanbhag et al. reported that shuffled data increased transaction abort rate as more cache lines were shared among the threads. Their experimental results show a big performance gap between sorted and shuffled datasets [28], which we also observe in our results.

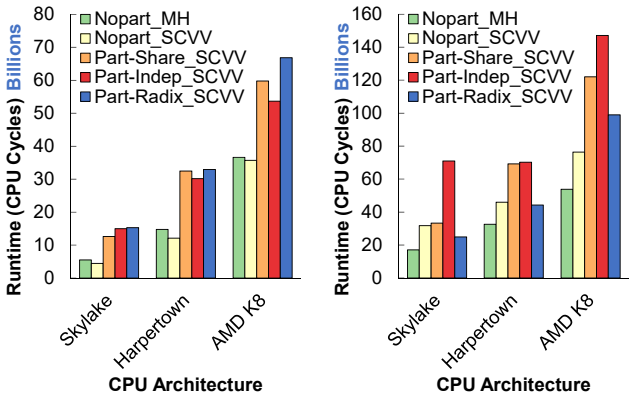
MH provides more consistent results as is indicated by the fact that the build times are relatively similar for shuffled and ordered datasets. As MH is based on Cuckoo hashing (which uses two different hash functions), its access pattern is not sequential. MH does not benefit from build table locality, but it guarantees that it will find a key in either one or two lookups.

6.7 CPU Architecture

Figure 10 depicts the total hash join runtimes we measured on three different hardware platforms. On all three hardware platforms, we run the experiments using identical source code and datasets. That is, we do not fine tune the algorithm to any particular architecture.

Our results indicate similar trends among the different architectures. Although we observe some minor variations, the fastest and slowest configurations are consistent. Despite large differences in CPU cache size (shown in Table 1), non-partitioning hash joins provide the fastest performance. A query planner with some knowledge of the data would be able to choose the fastest join variant.

The Part-Radix configuration performs relatively well on the shuffled dataset, but gradually loses ground going from Skylake to Harpertown and then AMD. As noted in [5] Radix hash join



(a) **Sequential** - Ordered (b) **Sequential** - Shuffled

Figure 10: Total join time on variable CPU architectures - 1:N Datasets - 8 threads

benefits from fine tuning the parameters to the hardware. This trend demonstrates that without parameter tuning, Radix hash join does not perform consistently.

Choosing the wrong join configuration can even result in modern hardware performing worse compared to old hardware. In Figure 10(b), the Part-Indep configuration performs similarly on Skylake and Harpertown. As noted in Section 6.4.3, Part-Indep may create an excessive number of private partitions. The combination of poor load balancing and unordered data access results in a performance penalty that is so severe, it undermines eight years of architectural improvements.

Shuffled datasets have the potential to thrash the cache and TLB. These datasets have a greater performance impact on the partitioned hash join variants. Non-partitioning hash joins provide faster and more predictable performance, regardless of the CPU architecture.

7 CONCLUSION

Hash joins are among the key techniques that enable efficient data access. However, their performance can be hindered when data is skewed and/or shuffled. We explored this issue on a set of 16 datasets that we have developed. We highlighted the importance of using a broad variety of synthetic datasets to mimic real-world applications. To our knowledge, no previous work has generated such a variety of datasets and analyzed their performance impact on hash joins.

We proposed modifications to the separate chaining based hash table (used in [8]) to deal with skewed data, and incorporated this into the hash join implementations used in their benchmark. We showed how the choice of hash table can improve performance with skewed datasets by more than three orders of magnitude when using our modified hash table compared to the prior implementation. With extensive experiments, we presented a performance evaluation of five hash join configurations.

In order to further improve performance, we introduced a novel hash table called Maple hash table. We have elaborated on how our hash table guarantees constant lookup cost, and described the many mechanisms we use to improve its insertion performance. We have shown that Maple hash table can further improve performance on shuffled datasets, and demonstrated speed-ups of up to 17.3×. Our study reinforces the case for more research in the area of hash joins on skewed or shuffled datasets, and the need for improved query optimizers that can choose a performant join configuration based on the data characteristics.

REFERENCES

[1] D. A. F. Alcantara. *Efficient Hash Tables on the GPU*. PhD thesis, University of California, Davis, 2011.
[2] Y. Arbitman, M. Naor, and G. Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. *ICALP*, pages 107–118, 2009.
[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *VLDBJ*, pages 85–96, 2013.
[4] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, pages 362–373. IEEE, 2013.
[5] Ç. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on modern processor architectures. *TKDE*, pages 1754–1766, 2015.
[6] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *VLDBJ*, pages 353–364, 2014.
[7] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *PODS*, pages 212–223. ACM, 2014.
[8] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, pages 37–48. ACM, 2011.
[9] T. P. P. Council. Tpc-h benchmark specification. *Published at http://www.tcp.org/hspec.html*, pages 592–603, 2008.
[10] A. Crolotte and A. Ghazal. Introducing Skew into the TPC-H Benchmark. In *TPCTC*, pages 137–145, 2012.
[11] B. Cutt and R. Lawrence. Improving join performance for skewed databases. In *CCECE*, pages 387–392. IEEE, 2008.
[12] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI 13*, pages 371–384, 2013.
[13] P. Garcia and H. F. Korth. Database hash-join algorithms on multithreaded computer architectures. In *CF '06*, pages 241–252. ACM, 2006.
[14] G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in microsoft sql server. In *VLDB*, pages 86–97, 1998.
[15] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Sigmod*, pages 243–252. ACM, 1994.
[16] H. Guiroux, R. Lachaize, and V. Quéma. Multicore locks: The case is not closed yet. In *USENIX ATC*, pages 649–662, 2016.
[17] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing load imbalance in real-world networked caches. In *HotNets*, page 8. ACM, 2014.
[18] T. Kejser. TPC-H Schema and Indexes. http://kejser.org/tpc-h-schema-and-indexes/, Jun 2014 (accessed June 16, 2017).
[19] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.
[20] 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. *VLDBJ*, pages 1378–1389, 2009.
[21] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. In *ESA*, pages 611–622, 2008.
[22] S. Kumar, J. Turner, and P. Crowley. Peacock hashing: Deterministic and updatable hashing for high performance networking. In *INFOCOM*, pages 101–105. IEEE, 2008.
[23] H. Lang, V. Leis, M. Albutiu, T. Neumann, and A. Kemper. Massively parallel numa-aware hash joins. In *IMDM*, pages 3–14, 2013.
[24] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys*, pages 27:1–27:14. ACM, 2014.
[25] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *TKDE*, pages 709–730, 2002.
[26] R. Pagh and F. F. Rodler. Cuckoo hashing. In *ESA*, pages 121–133. Springer, 2001.
[27] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *VLDBJ*, pages 96–107, 2015.
[28] A. Shanbhag, H. Pirk, and S. Madden. Locality-adaptive parallel hash joins using hardware transactional memory. In *IMDM*, 2016.
[29] C. Silverstein. Google sparsehash. *https://github.com/sparsehash/sparsehash*, 2005.