Re-evaluating In-Memory Parallel Hash Join Designs

Zhidong Guo[1], **Ye Yuan**[1]
[1]Information Networking Institute, Carnegie Mellon University
`{zhidongg, yeyuan3}@andrew.cmu.edu`

Abstract

Our project investigates the efficiency of in-memory parallel hash join variants, a critical component in relational database management systems, particularly when handling large datasets. We explore two main variants of the hash join algorithm: Shared Hash Join and Partitioned Hash Join. Both variants are examined under two task scheduling strategies - static and dynamic - and their performance is analyzed across multiple workloads with varying degrees of skewness. The implementations leverage the unique properties of Rust, providing a robust platform for detailed empirical evaluation and theoretical analysis. Our study systematically compares the two approaches in terms of execution time, synchronization, and cache utilization. Our parallel implementation reaches $\times 6.5$ speedup with 8 cores. The results indicate that while shared hash joins excel in scenarios with high data skew due to improved cache locality, partitioned hash joins perform better in uniformly distributed datasets owing to reduced cache misses. The findings suggest that selecting an optimal hash join strategy is crucial for enhancing query performance in modern relational databases, considering the specific characteristics of the workload and data distribution. Through rigorous analysis, this project contributes to a deeper understanding of parallel hash join techniques, offering insights that can guide the implementation of more efficient and adaptable database systems. Our code is available at https://github.com/cmu-15618-team/parallel-hash-join.

1 Introduction

The join operation is a cornerstone in the domain of relational database management systems (DBMS) [2]. It facilitates combining data across multiple tables, so that the users can organize their data in multiple normalized relations which more accurately reflects the data model in reality, eliminates duplicated data, ensures consistency, and so forth. The efficiency of join operations is paramount as they are integral to nearly every SQL query executed within a DBMS. An inefficient join operation can significantly degrade the performance of a database system, leading to slow query responses and decreased user satisfaction.

Among the plethora of algorithms devised to perform join operations, the hash join algorithm stands out due to its conceptual simplicity and its linear time complexity with respect to the number of tuples in both relations. This efficiency makes hash joins particularly appealing in scenarios involving large datasets where join operations can become a bottleneck.

Hash joins divide the join process into two distinct phases: the build phase, where a hash table is created from the smaller of the two relations, and the probe phase, where the larger relation is scanned and matched against the hash table. This two-step process allows hash joins to quickly identify matching tuples, thereby speeding up the join process. However, the performance of hash joins can be significantly influenced by the size of the relations, the distribution of data, and the available memory, which affects how effectively the hash tables can be managed and accessed during the join.

Project of Spring 2024 15-618 parallel programming course at Carnegie Mellon University. Do not distribute.

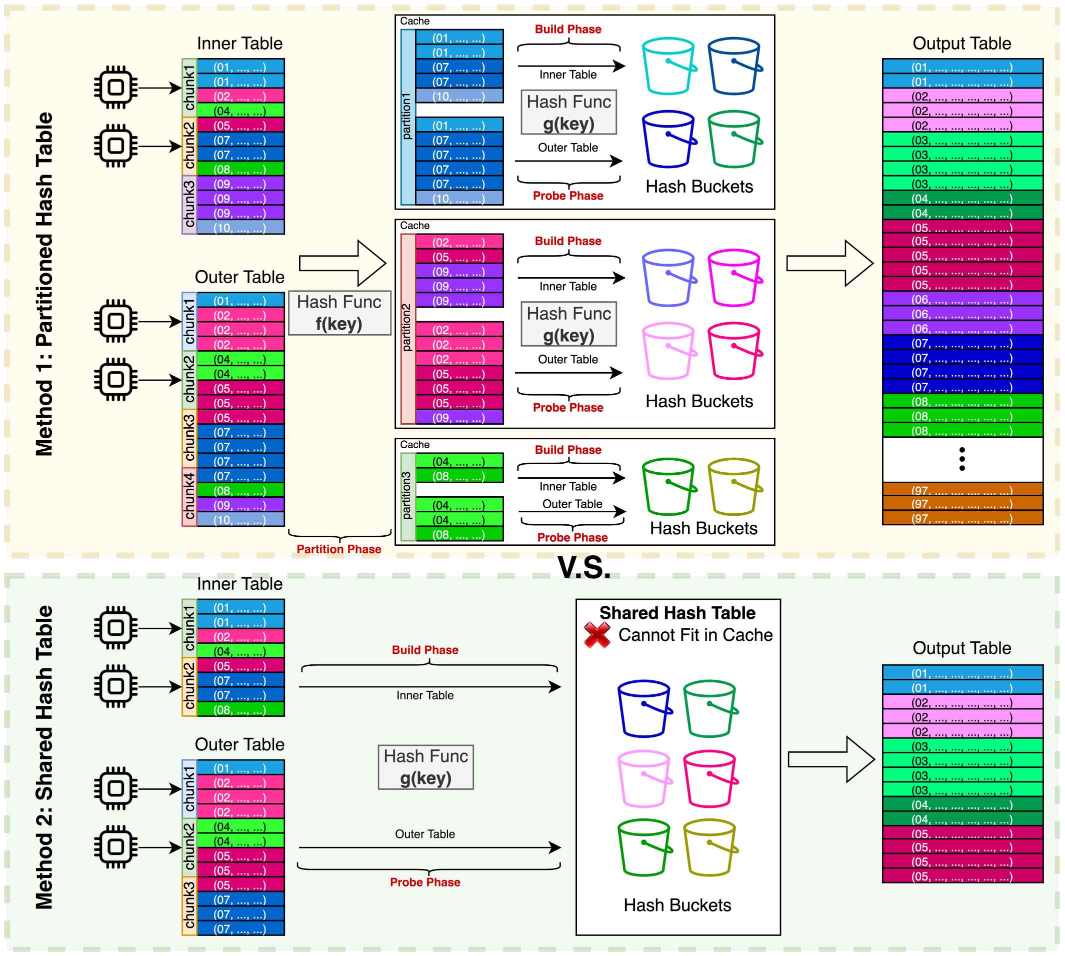


Figure 1: Overview of two parallelism patterns.

Given the critical importance of join operations in relational databases and the prevalent use of the hash join algorithm, it is essential to explore techniques that enhance its efficiency and adaptability. This includes investigating different partition strategies such as building a shared hash table versus multiple hash tables, one for each partition, and comparing task scheduling strategies including static and dynamic scheduling. These considerations can help optimize hash joins under various data distributions, potentially leading to substantial improvements in query performance.

This project aims to delve into these optimizations, presenting a comprehensive analysis comparing shared versus partitioned hash tables and static versus dynamic scheduling approaches. Through empirical evaluation and theoretical analysis, we seek to uncover insights that can guide the implementation of hash joins in modern relational DBMSs. Our contributions are:

- Implementation of two variants of hash joins. The first variant is **Shared Hash Join** where all threads build a shared hash table. The second variant is **Partitioned Hash Join**, where the input relations are split into partitions on which joins are performed locally. Both variants support static and dynamic task scheduling.
- Implementation of the benchmark infrastructure to evaluate the performance of various parallelism patterns under three types of workloads with varying skewness.
- Comprehensive evaluations, including a phase-by-phase execution time comparison, synchronization analysis, and cache analysis. The extensive results from these evaluations provide insights into the performance bottlenecks, the effects of various optimizations, the applicability of both hash join variants, and the details to notice during implementation.

2 Methodology

A hash-join algorithm works on two input relations, naming inner table (R) and outer table (S)[1]. We assume that $|R| < |S|$ and the join key is the first column of both tables. Typically, a hash join algorithm consists of three phases: partition (optional), build, and probe. The partition phase is optional and is used to partition the input tables R and S into multiple partitions. The build phase is used to build a hash table from the inner table R. The probe phase is used to probe the hash table with the outer table S to find the matching tuples. Figure 1 shows the overview of hash join algorithm w/ partitioning and w/o partitioning.

Algorithm 1 Parallel Hash Join Process

```
Require: Relations $R$ (inner table) and $S$ (outer table), Number of partitions $n$, Number of threads $t$
Ensure: Joined tuples from $R$ and $S$
  function PARALLELHASHJOIN($R, S, n, t$)
      if partitioning is enabled then
          $R_1, R_2, \ldots, R_n \leftarrow$ PARTITION($R, n$)
          $S_1, S_2, \ldots, S_n \leftarrow$ PARTITION($S, n$)
      else
          $R_1 \leftarrow R$
          $S_1 \leftarrow S$
      end if
      Initialize hash tables for each partition
      BUILDPHASE($R_1, R_2, \ldots, R_n, t$)
      Initialize output buffers for each thread
      PROBEPHASE($S_1, S_2, \ldots, S_n, t$)
      Collect and concatenate outputs from all threads
      return all joined tuples
  end function
  function PARTITION($R, n$)
      Divide $R$ into $n$ partitions using hash function $h_1(\cdot)$ based on join keys
      return partitions
  end function
  function BUILDPHASE($Partitions, t$)
      for each thread $i = 1$ to $t$ do
          Assign partitions to thread $i$
          Build hash buckets with hash function $h_2(\cdot)$ for assigned partitions
      end for
  end function
  function PROBEPHASE($Partitions, t$)
      for each thread $i = 1$ to $t$ do
          Assign partitions to thread $i$
          for each tuple $s$ in assigned partition of $S$ do
              Probe hash tables with hash function $h_2(\cdot)$ using join key from $s$
              Store matching tuples in thread's output buffer
          end for
      end for
  end function
```

2.1 Partition Phase

The partition phase, while optional, plays a crucial role in optimizing the processing of large input tables R and S. During this phase, the tables are divided into n partitions, denoted as $R_1, R_2, \ldots, R_n$ and $S_1, S_2, \ldots, S_n$, respectively, where n represents the total number of partitions. Each partition is designed to be a subset of the original table, ideally sized to fit within the cache to minimize cache misses.

[1]We may use R and S interchangeably with inner and outer table in the following sections.

The partitioning is accomplished using a hash function, $h_1(\cdot)$, applied to the join key, k. This function distributes the tuples across the partitions based on the computed hash values, aiming to balance the load and reduce the likelihood of cache misses during the join operation. The partition function is defined as follows:

$$P(k) = h_1(k) \mod n \tag{1}$$

Where $P(k)$ represents the partition number to which the tuple with join key k is assigned. This equation ensures that each tuple is assigned to a partition, while providing a straightforward yet effective mechanism to distribute the data evenly, assuming the join key values are uniformly distributed across the range of possible values.

2.2 Build Phase

The build phase is a critical step in hash join operations and its execution depends on whether the input relations were partitioned. In the absence of partitioning, all threads are collectively assigned to work on the entire relation R. Conversely, if partitioning was employed, the assignment of partitions to threads is either in a static way or a dynamic way.

Static assignment. Each thread i is responsible for managing the partitions $R_i, R_{i+t}, R_{i+2t}, \ldots$, where t represents the total number of threads. Mathematically, this can be described by the assignment function:

$$\text{Thread } i \text{ works on } R_{i+kt} \quad \text{for } k = 0, 1, 2, \ldots \tag{2}$$

For instance, with 8 threads, thread 0 would handle partitions $R_0, R_8, R_{16}, \ldots$ and so on.

Dynamic assignment. Each thread i, where $i \in \{0, 1, \ldots, t-1\}$, is initially assigned a specific partition R_p from the set of partitions $\{R_0, R_1, \ldots, R_{t-1}\}$. Or more precisely, the initial assignment can be expressed as $R_{p=i \mod t}$ for each thread i. After a thread completes processing its assigned partition, it retrieves the next unprocessed partition from the sequence, R_x, where $x \in \{t, t+1, \ldots, n-1\}$. This assignment process continues in a first-in-first-serve fashion until all partitions, represented as R_x for $x \in [0, n)$, have been processed.

Upon assignment, each thread initializes an empty hash table for each of its designated partitions. To optimize cache utilization, the hash table is structured so that each bucket can comfortably fit within a few cache lines. This design minimizes cache misses during subsequent operations. Each thread processes every tuple r in its assigned partition by extracting the join key k_r, and hashes this key using a secondary hash function $h_2(\cdot)$, distinct from the hash function used in the partitioning phase:

$$b_p(k_r) = h_2(k_r) \mod m \tag{3}$$

where $b_p(k_r)$ denotes the bucket of partition p that the tuple r is assigned to, and m is the number of hash buckets of each partition. Each tuple r is then appended to the corresponding hash bucket. If a particular bucket does not yet exist, it is created dynamically.

In scenarios where partitioning is not applied, all threads share a common hash table. To ensure thread safety amidst concurrent writes, each hash bucket is protected by a synchronization mechanism, such as a latch or mutex, as implemented using a Rust Mutex in our case. This mutual exclusion ensures data integrity but may introduce overhead due to contention.

The build phase is considered complete once all threads have finished processing their respective partitions or the entire relation R. The completion condition can be formally stated as:

$$\forall i \in \{0, 1, \ldots, t-1\}, \quad \text{Thread } i \text{ has processed } R_{i+kt}, \quad k \in \mathbb{N} \tag{4}$$

where $\mathbb{N}$ includes all non-negative integers appropriate to the number of partitions or the size of R.

2.3 Probe Phase

During the probe phase, thread assignments are similar to those in the build phase. In cases where no partitioning is employed, all threads collectively work on the entire relation S. This can be either static or dynamic, as described in Section 2.2. Conversely, if partitioning has been executed, either each thread i is responsible for the partitions $S_i, S_{i+t}, S_{i+2t}, \ldots$ when the assignment is static, or each thread dynamically retrieves a partition sequentially from the inner table S.

Within this phase, each thread iterates over every tuple $s \in S_i$, extracting the join key k_s. For each tuple s, the thread queries the hash table using the hash function $h_2(\cdot)$:

$$b_p(k_s) = h_2(k_s) \mod m \tag{5}$$

where a, b, and p are parameters ensuring uniform distribution of keys. The thread then retrieves the corresponding bucket $b_p(k_s)$ from the hash table and checks each tuple r within the bucket:

$$\text{Check if } k_r = k_s \quad \forall r \in h(k_s) \tag{6}$$

This comparison is crucial to exclude false positives due to hash collisions.

Upon finding a match, i.e., when $k_r = k_s$, the tuples r and s are joined to produce an output tuple (r, s) and are appended to an output buffer exclusive to each thread.

2.4 Rust Implementation Details

Mutex Instead of using `std::sync::Mutex`, we chose `parking_lot::Mutex` due to its better performance and memory footprint - one `parking_lot::Mutex` object only requires 1 byte of storage space. [1] For the remainder of this section, we will refer to it as `Mutex`.

Hash Function We picked xxh3 with different seeds as $h_1(\cdot)$ and $h_2(\cdot)$ based on the benchmark results from [4] revealing that it is the fastest hash function without quality problems.

Multi-threading Rayon is a Rust multi-threading library that provides handy functionalities including thread pools and dynamic work stealing.

Partition buffer A partition is essentially a consecutive buffer of tuples, implemented with `boxcar::Vec`, a lock-free vector implementation. We chose `boxcar::Vec` over `Mutex<std::Vec>` because the former provides better write performance at the cost of slightly worse read performance. Table 1 shows the results of a micro-benchmark where 8 threads each push a million `i32` values into the two data structures. After all the threads finish pushing the values, they iterate through the vectors in parallel. Since all the tuples are written to and read from the partition buffer exactly once, we should pick the data structure with the smaller total time.

Table 1: Micro-benchmark results comparing the performance of `boxcar::Vec` and `Mutex<std::Vec>`.

	`boxcar::Vec`	`Mutex<std::Vec>`
Write (Push)	485ms	905ms
Read (Iterate)	16ms	5ms
Total	501ms	910ms

Hash Bucket We implemented the hash bucket with `Mutex<std::Vec>` because a hash bucket is iterated over multiple times during the probe phase. The average read-to-write ratio is $|S|/|R|$, which is typically greater than 10. Therefore, we need to maximize the read throughput with `Mutex<std::Vec>`.

3 Experiment

We have implemented the in-memory hash join algorithms proposed in Section 2 and conducted experiments to evaluate their performance. The program first generates two tables in memory, namely R or inner table and S or outer table. The sizes of the tables are determined by the input parameters, including `inner_tuple_num` and `outer_ratio`. Unless specified, the default value of input parameters is listed in Table 2. The program then performs the hash join operation on the two tables and records the execution time of each phase introduced in Section 2. The program is implemented in Rust, and compiled with `rustc 1.77.2`. The program is executed multiple times for each configuration, and the average execution time is reported.

Parameter Name	Default Value	Explanation
`inner_tuple_num`	16,000,000	Number of tuples in the inner relation.
`outer_ratio`	16	The ratio of the number of tuples in the outer relation to the inner relation.
`batch_size`	100,000	Number of tuples in each batch.
`partition_num`	4096	Number of partitions.
`bucket_num`	1,048,576	Total number of buckets in the hash table(s).
`threads`	8	Number of threads to use.

Table 2: Default Parameters Explanation

3.1 Experimental Setup

3.1.1 Hardware Specification

The experiments are conducted on the platform with the hardware specification listed in Table 3.

	Pittsburgh Supercomputing Center (PSC)
CPU	AMD EPYC 7742
Cores	64
Cache Size	256MB L3
Memory	256GB

Table 3: Hardware Specification

3.1.2 Dataset

The tuples in both tables are `(key, payload)` pairs, where both `key` and `payload` are 8 bytes long. We chose to represent the payload with 8 bytes to simulate `(key, record_id)` a column store with late materialization. We also assume that the join is a primary-foreign key join, which means the key in R is the primary key as if generated through `AUTO INCREMENT`, while the keys in S are generated in memory following the Zipfian distribution [3]. The Zipfian distribution is a power-law probability distribution that is often used to model the distribution of data in real-world applications. The Zipfian distribution is defined as follows:

$$P(k) = \frac{1/k^a}{\sum_{n=1}^{N} 1/n^a} \tag{7}$$

where $P(k)$ is the probability of the k-th element, a is the skew parameter, and N is the total number of elements. The skew parameter a determines the skewness of the distribution. A larger a value results in a more skewed distribution, which means that a few join keys appear more frequently than others. Table 4 shows the statistics of the hash buckets size distribution for different skew parameters. In the following experiments, the uniform setting uses $a = 0.0$, the low skew setting uses $a = 1.05$, and the high skew setting uses $a = 1.25$. It is important to note that **even if the keys are uniform, the workload is still not perfectly balanced**.

Intuitively, the workload becomes more imbalanced as the skew parameter a increases. The mean and standard deviation of the bucket size distribution under different skew parameters a are shown in Table 4. The mean bucket size remains constant at 244.14, while the standard deviation increases as the skew parameter a increases. The standard deviation measures the spread of the bucket size distribution. A higher standard deviation indicates a more imbalanced workload, which can lead to performance degradation in parallel hash join algorithms.

3.2 Speedup of Parallelism

This section analyzes the overall and per-phase speedup of shared hash join compared with sequential hash join under different workloads and task scheduling strategies. The primary purpose of this

Table 4: The mean and standard deviation of the bucket size distribution under different skew parameters a. It can be observed that the workload becomes more imbalanced as the skew parameter a increases.

alpha	mean	std
0 (uniform)	244.14	64.38
0.1 (low skew)	244.14	64.78
0.2	244.14	66.38
0.3	244.14	70.85
0.4	244.14	83.96
0.5	244.14	130.54
0.6	244.14	306.06
0.7	244.14	918.36
0.8	244.14	2838.32
0.9	244.14	7984.42
1	244.14	18741.04
1.1	244.17	35250.42
1.2	245.67	54592.92
1.3	267.56	76501.34
1.4	367.25	110206.59
1.5	628.01	167823.65
1.6	1174.3	258033.9
1.7	2222.55	389725.16
1.8	4053.52	567971.11
1.9	7079.25	799766.48
2 (high skew)	11823.39	1090601.67

section is to gain insights into the workload characteristics of the build and probe phase. These insights apply to both parallel hash join variants, so we omit the analysis for partitioned hash join.

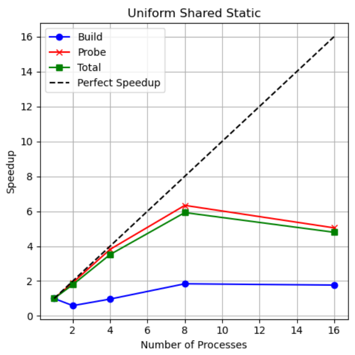
A detailed inspection of the figure reveals that the speedup escalates as the number of threads increases, provided that this number does not surpass the maximal hardware parallelism. Beyond this threshold, a diminution in performance is noted, predominantly attributable to the overhead associated with context switching.

Specifically, the probe phase of the operation achieves a maximal speedup of approximately 7X when utilizing 8 threads under uniform data distribution and dynamic task scheduling. In this configuration, a total speedup of 6.5 is attained, underscoring the efficiency of parallel processing for the probe phase over the build phase. The significant speedup is largely due to the **probe phase's read-only data access pattern, which means no synchronization is required**. Conversely, **the build phase involves frequent locking and unlocking of hash buckets** incurring substantial synchronization overhead. This explains the performance drop of the build phase with 2 threads in all cases, where the synchronization overhead overshadows the benefit of parallelism.

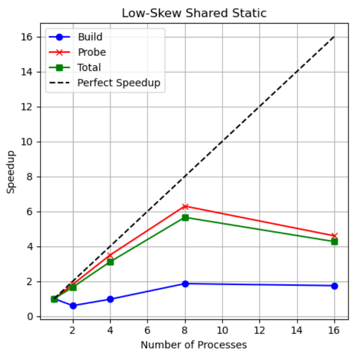
In terms of scheduling strategy, dynamic scheduling achieved comparable or better speedup under all workloads, due to imbalanced workloads and Rayon's efficient implementation of dynamic scheduling.

3.3 Comparison of Parallelism Patterns

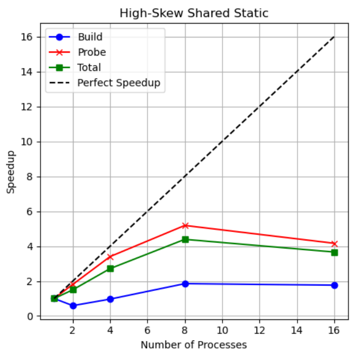
This section compares the performance of varying parallelism patterns, focusing on static versus dynamic task scheduling strategies and the use of shared versus partitioned hash tables under diverse workload distributions. The investigation centers on how these configurations influence the execution times across different phases of the hash join algorithm. The results of our empirical evaluations are shown in Figure 3. For partitioned hash join, we choose the partition size which gives the optimal performance.



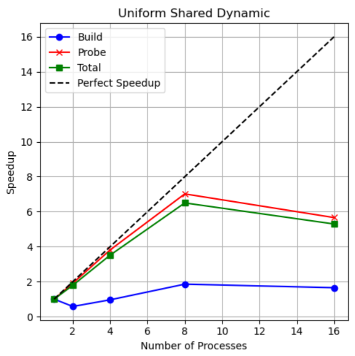
(a) Uniform Shared Static



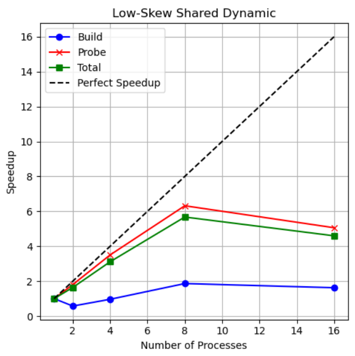
(c) Low Skew Shared Static



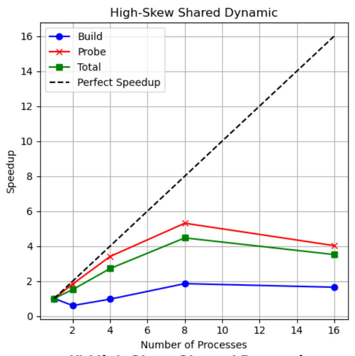
(e) High Skew Shared Static



(b) Uniform Shared Dynamic

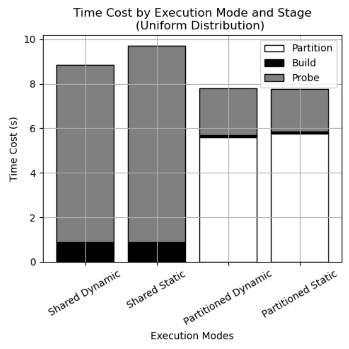


(d) Low Skew Shared Dynamic

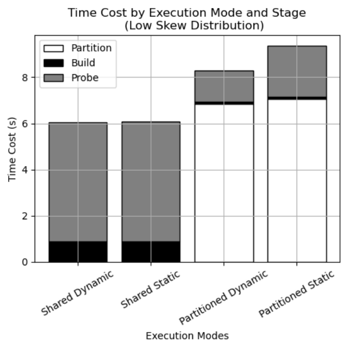


(f) High Skew Shared Dynamic

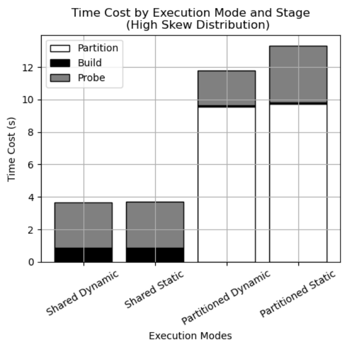
Figure 2: Illustration of the speedup achieved by the parallel hash join algorithm across various skew patterns and task scheduling strategies. Notable observations include an increase in speedup commensurate with the number of threads, up to the limit imposed by the number of physical processor cores. Dynamic scheduling achieved comparable or better speedup under all workloads, underscoring Rayon's efficient implementation of work stealing.



(a) Uniform Distribution



(b) Low Skew Distribution



(c) High Skew Distribution

Figure 3: Comparative performance analysis of various parallelism patterns across differing workload distributions. Key findings include that partitioned hash join performs better in uniform workloads, whereas shared buffers excel under skewed workloads. Additionally, dynamic task scheduling is vital for partitioned hash join under skewed workloads.

3.3.1 Impact of Workload Distribution

Partitioned hash join outperforms shared hash join under uniform data distribution. This superiority can be attributed to the lower cache miss rate achieved by partitioning the entire dataset into smaller partitions that fit into the cache. The non-trivial partitioning overhead is offset by the significantly faster build and probe phases.

However, partitioned hash join performs worse than shared hash join under skewed distributions. The reasons are threefold. First, the contention over hot partition buffers gets worse as data becomes more skewed, which explains the increasing time of the partition phase. Second, the workload balance of shared hash join is agnostic of data distribution, since we are simply assigning tuples to threads, and each tuple takes roughly the same amount of time to probe. Third, skewed data translates into better temporal cache locality for shared hash join, which explains the decreased probe phase time as data gets more skewed. By contrast, in partitioned hash join, the partitions already fit in the cache so locality is not improved.

3.3.2 Impact of Scheduling Strategy

Looking at Partitioned Dynamic versus Partitioned Shared in Figure 3 (b) and (c), we can conclude that Dynamic task scheduling plays a pivotal role in balancing the workload across threads during the probe phase. In addition, dynamic scheduling performs equally well or better than static scheduling even for shared hash join and partitioned hash join under uniform distribution, highlighting the superior implementation of Rayon.

3.4 Cache Analysis

This section elaborates on the impact of cache utilization as influenced by the parallelism patterns discussed in the previous sections through performance counters. Two key observations can be made from Table 5.

Table 5: Detailed Cache Miss Rate Analysis across Different Parallelism Patterns. The table presents data for various configurations, designated by modes such as `USD` (Uniform Shared Dynamic), `UPD` (Uniform Partitioned Dynamic), `LSD` (Low Skew Shared Dynamic), `LPD` (Low Skew Partitioned Dynamic), `HSD` (High Skew Shared Dynamic), and `HPD` (High Skew Partitioned Dynamic). The metrics include the number of cache misses, the number of cache references, and the cache miss rates.

configuration	`USD`	`LSD`	`HSD`
cache miss num	2,513,975,929	827,780,779	355,003,945
cache ref num	4,321,685,631	2,236,064,733	1,287,520,618
cache miss rate	**58.17%**	**37.01%**	**27.57%**
configuration	`UPD`	`LPD`	`HPD`
cache miss num	1,201,097,267	1,142,299,850	1,315,174,783
cache ref num	5,693,788,141	4,447,116,933	5,037,420,108
cache miss rate	**21.09%**	**25.68%**	**26.10%**

First, partitioned hash join exhibits an overall low cache miss rate. This is in line with the design principle of partitioned hash join that partitions should fit entirely into the cache. The cache misses happen during the partition phase and at the beginning of the build and probe phase when the cache is cold.

Second, the cache miss rate of shared hash join decreases dramatically as data skewness increases. Skewed data means hot values occur more often, which translates into increased temporal cache locality.

3.5 Synchronization Overhead Analysis

The synchronization overhead in parallel processing environments is a critical performance factor, especially in the context of database operations such as hash joins. We use CPU cycle per output tuple to measure the synchronization overhead as presented in Table 6.

Table 6: Quantitative analysis of CPU cycle per output tuple across various hash join configurations, highlighting the synchronization overhead associated with different data distributions, partition strategies, and scheduling strategies. Each configuration (e.g., `USD` for Uniform Shared Dynamic) represents a specific combination of data distribution (Uniform, Low skew, High skew) and strategies (Shared vs. Partitioned, Dynamic vs. Static). Lower CPU cycles per output tuple indicate lower synchronization overhead. Since the cycles are composed of synchronization overhead and the actual computation, we measure the baseline from sequential hash join, denoted UQ, LQ or HQ.

configuration	`UQ`	`USD`	`USS`	`UPD`	`UPS`
CPU cycle per output tuple	739	802	792	729	646
configuration	`LQ`	`LSD`	`LSS`	`LPD`	`LPS`
CPU cycle per output tuple	599	667	657	834	796
configuration	`HQ`	`HSD`	`HSS`	`HPD`	`HPS`
CPU cycle per output tuple	364	414	403	613	600

3.5.1 Uniform Data Distribution

In uniform data distributions, the results indicate that partitioned configurations (both dynamic and static) tend to perform better than their shared counterparts. Specifically, the Uniform Partitioned Dynamic (`UPS`) configuration requires the fewest CPU cycles per output tuple at 646 cycles, compared to 792 cycles (-146 cycles) in the Uniform Shared Dynamic (`USS`). This suggests that partitioning reduces contention among threads accessing shared data structures, hence lowering synchronization overhead.

3.5.2 Low Skew Data Distribution

For low skew distributions, the pattern reverses. The Low Skew Shared Dynamic (`LSD`) configuration shows better performance (667 CPU cycles per output tuple) compared to the Low Skew Partitioned Dynamic (`LPD`), which requires 834 cycles. This reduced efficiency in the partitioned setup stems from increased contention within partitions: as skew increases, the likelihood of uneven data distribution across partitions grows. This can lead to some partitions being significantly more loaded than others, causing higher contention during the partition phase.

3.5.3 High Skew Data Distribution

The performance metrics for high skew distribution demonstrate distinct differences between shared and partitioned configurations. The High Skew Shared Dynamic (`HSD`) configuration, requiring 414 CPU cycles per output tuple, outperforms the High Skew Partitioned Dynamic (`HPD`) configuration, which consumes 613 CPU cycles per output tuple. Similar to the analysis in Section 3.5.2, this performance degradation could be attributed to the imbalance of workload across partitions, which may produce larger partitions and lead to more contention within those partitions.

3.5.4 Dynamic vs. Static Scheduling

Across all data distributions, dynamic configurations generally incur higher CPU cycles per output tuple compared to static ones due to the overhead of dynamically assigning tasks based on the current workload. However, the overhead is negligible compared with the benefit of a balanced workload as shown in Figure 3.

3.5.5 Implications and Insights

The implications of these results for database system design are significant. For uniform or low skew workloads, partitioned hash join might be more advantageous because of the lower synchronization overhead. Conversely, in high skew workloads, shared configurations may be preferable.

To optimize performance, database systems should consider dynamically switching between shared and partitioned configurations based on statistics. Additionally, optimizing the dynamic task scheduling algorithms to reduce coordination overhead is a key aspect of an efficient implementation of partitioned hash join.

3.6 Effect of Partition Number on Performance

Obtaining the optimal number of partitions is critical for maximizing the performance of partitioned hash join, since the choice of partition number significantly affects the cache miss rates. In this section, we use the Partitioned Dynamic configuration with 8 threads and vary the partition number to see how it affects the performance.

3.6.1 Theoretical Background

The number of partitions influences cache performance in two ways. First, since the amount of computation is proportional to the partition size, it's critical that even the largest partition should fit into the cache. Increasing the number of partitions potentially reduces the size of the largest partition, which makes it easier to fit it into the cache, thereby **decreasing the cache capacity miss rate**. However, as the partition number continues to increase, the average size of each partition decreases, leading to a larger number of hash buckets, each containing fewer tuples (the number of hash buckets per partition is fixed). This scenario reduces cache effectiveness due to **increased cold misses**, as new data must frequently be loaded from the main memory to the cache.

3.6.2 Empirical Analysis

Empirical observations, as illustrated in Figures 4 and 5, show a non-linear relationship between partition number and performance.

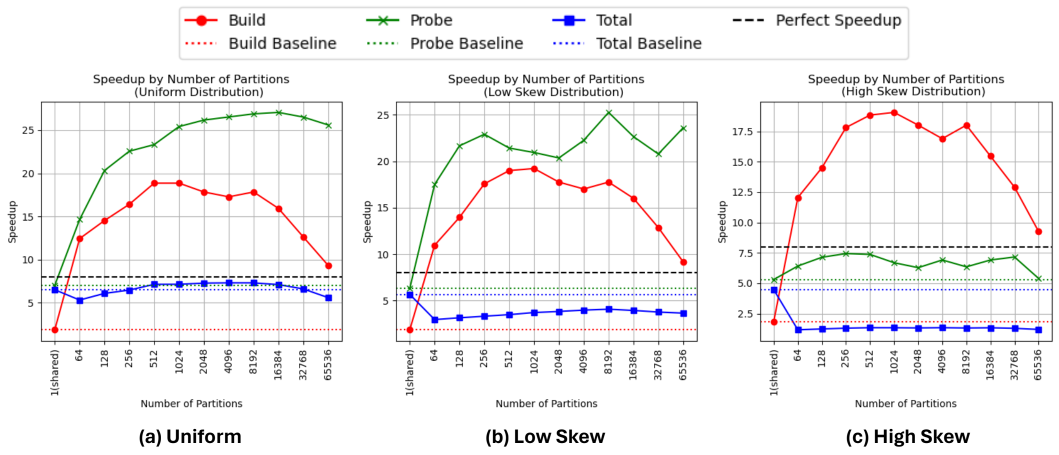
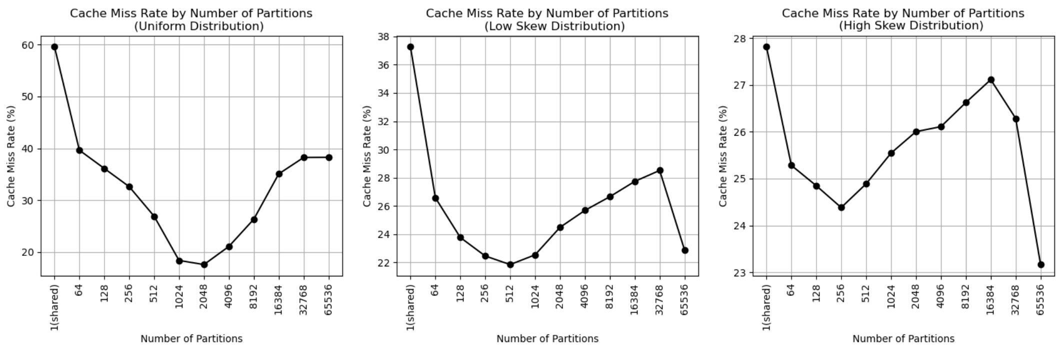


Figure 4: Illustration of performance variation with different partition numbers, showing an initial improvement followed by a decline, correlated with cache miss rate changes in Figure 5. The baselines are the speedup of shared dynamic hash join under 8 threads. The total speedup is significantly lower than that of the build and probe phase because we take partition overhead into account, as illustrated in Figure 6.

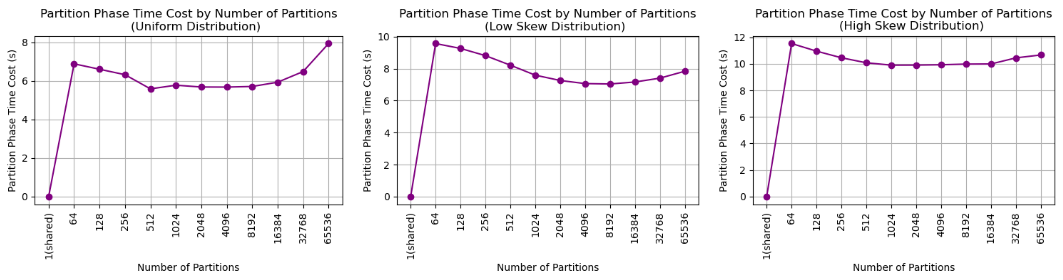
Taking uniform distribution as an example, we first analyze the relationship between the build/probe phase speedup and cache miss rate. The build phase speedup reaches the optimal point at 1024 and 2048, while the probe phase speedup increases quickly before 1024 and remains relatively stable after that. The cache miss rate shows an opposite trend, where the lowest point occurs in 2048, and either increasing or decreasing the partition number will increase the miss rate. As explained, the upturn after 2048 is attributed to the increase in code misses which dwarfs the benefit of reduced capacity misses, and the trend is reversed on the other side.

Next, we can see from Figure 6 that increased partition number leads to lower partitioning overhead due to reduced contention, which explains why the optimal total speedup point for Low Skew occurs at 8192, where the build and probe phase speedup has already taken a downturn.



(a) Uniform **(b) Low Skew** **(c) High Skew**

Figure 5: Cache miss rates as a function of partition number, highlighting the initial decrease and subsequent increase, explained by the interplay of reduced capacity misses and increased cold misses.



(a) Uniform **(b) Low Skew** **(c) High Skew**

Figure 6: Partition phase time cost vs. number of partitions. Note that shared hash table (when the partition number is 1) does not have partition phase, so the time cost is 0.

This analysis highlights the importance of selecting an appropriate partition number to balance the positive effect of reduced capacity misses against the negative impact of increased cold misses. In addition, one should avoid choosing a very small partition number as it increases partition overhead. The optimal partition number, therefore, strikes a balance between these factors maximizing cache efficiency and minimizing synchronization overhead.

4 Conclusion

In summary, our experiments have revealed the key takeaways listed as follows.

1. Shared hash join performs better when data distribution is skewed because a) workload balance is unaffected by data distribution, b) skewness improves cache temporal locality, c) there's no partitioning required.
2. Partitioned hash join performs well when the data distribution is relatively uniform due to the high utilization of the cache. However, as the data becomes more skewed, the synchronization overhead in the partition phase and workload imbalance overshadow the benefit of cache.
3. Database systems should enable their query optimizers to choose between these methods based on statistics.
4. A high-quality dynamic task scheduling algorithm is critical to mitigate the effect of workload imbalance for partitioned hash join.
5. Partition number is another critical factor to partitioned hash join's performance. A smaller number of partitions leads to higher partition overhead and increased cache capacity miss, while a larger number of partitions leads to increased cache cold misses. We need to find a sweet spot in between.

6. The probe phase is much more computationally heavy than the build phase. Therefore, the hash bucket's data structure should be optimized for read performance.

References

[1] Amanieu. parking_lot: Compact and efficient synchronization primitives for Rust. `https://github.com/Amanieu/parking_lot`. Accessed: 2024-05-04.

[2] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 37–48, New York, NY, USA, 2011. Association for Computing Machinery.

[3] Mark EJ Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[4] R. Urban. SMhasher. `https://github.com/rurban/smhasher`. Accessed: 2024-05-04.

A Contribution

A.1 Effort breakdown

- **Rust implementation:** Zhidong Guo 60%, Ye Yuan 40%
- **Experiment:** Zhidong Guo 40%, Ye Yuan 60%
- **Report, Poster and Logistics:** Zhidong Guo 50%, Ye Yuan 50%

A.2 Credit division

Since the two author contribute equally to the project, the total credits may divide into 50%-50%.