Reevaluating In-Memory Parallel Hash Join Designs

Zhidong (David) Guo, Ye (Ethan) Yuan

{zhidongg, yeyuan3}@andrew.cmu.edu

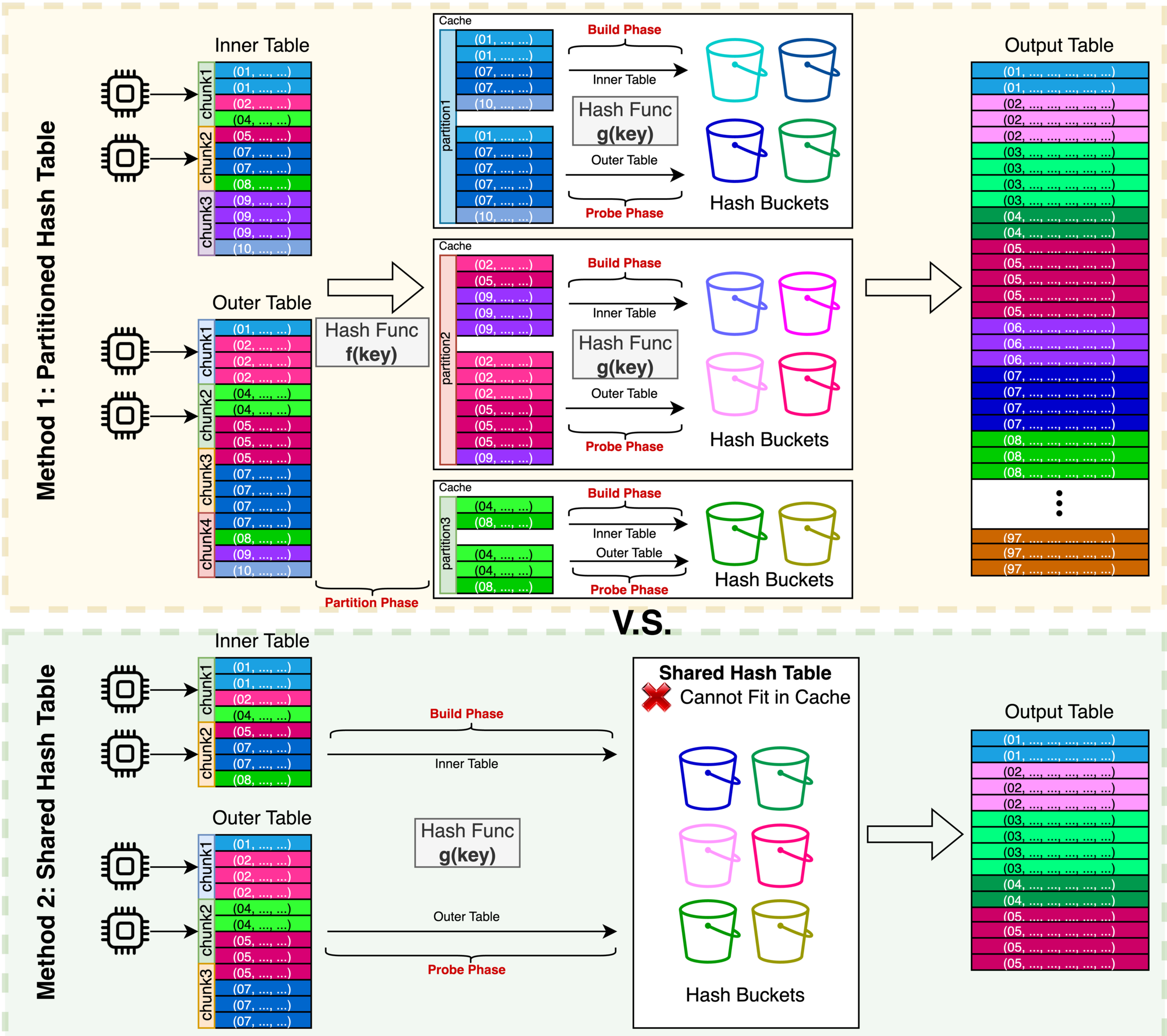
Introduction

Join operation is arguably the most important operation in relational DBMS. It offers the ability to combine two or more relations (tables) together so that the users can organize their data in multiple relations which more accurately reflects the data model in reality, eliminates duplicated data, ensures consistency, and so forth. It is also one of the most crucial operations in terms of performance because join appears in virtually every SQL query, and a bad implementation of join can be devastatingly slow. Among various join algorithms, hash join has been one of the most well-studied and widely-adopted algorithms due to its conceptual simplicity and its linear time complexity with respect to the number of pages in both relations (in a disk-oriented context).

Methodology

Shared vs. Partitioned Hash Tables (see the figure on the right)

- **Shared Hash Table:** Utilizes a single hash table accessible by all threads during build and probe phase.
- **Partitioned Hash Table:** First divide the inner and outer table into partitions (partition phase), then threads pick partitions and do the build and probe phase.



Experiment Setup

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.

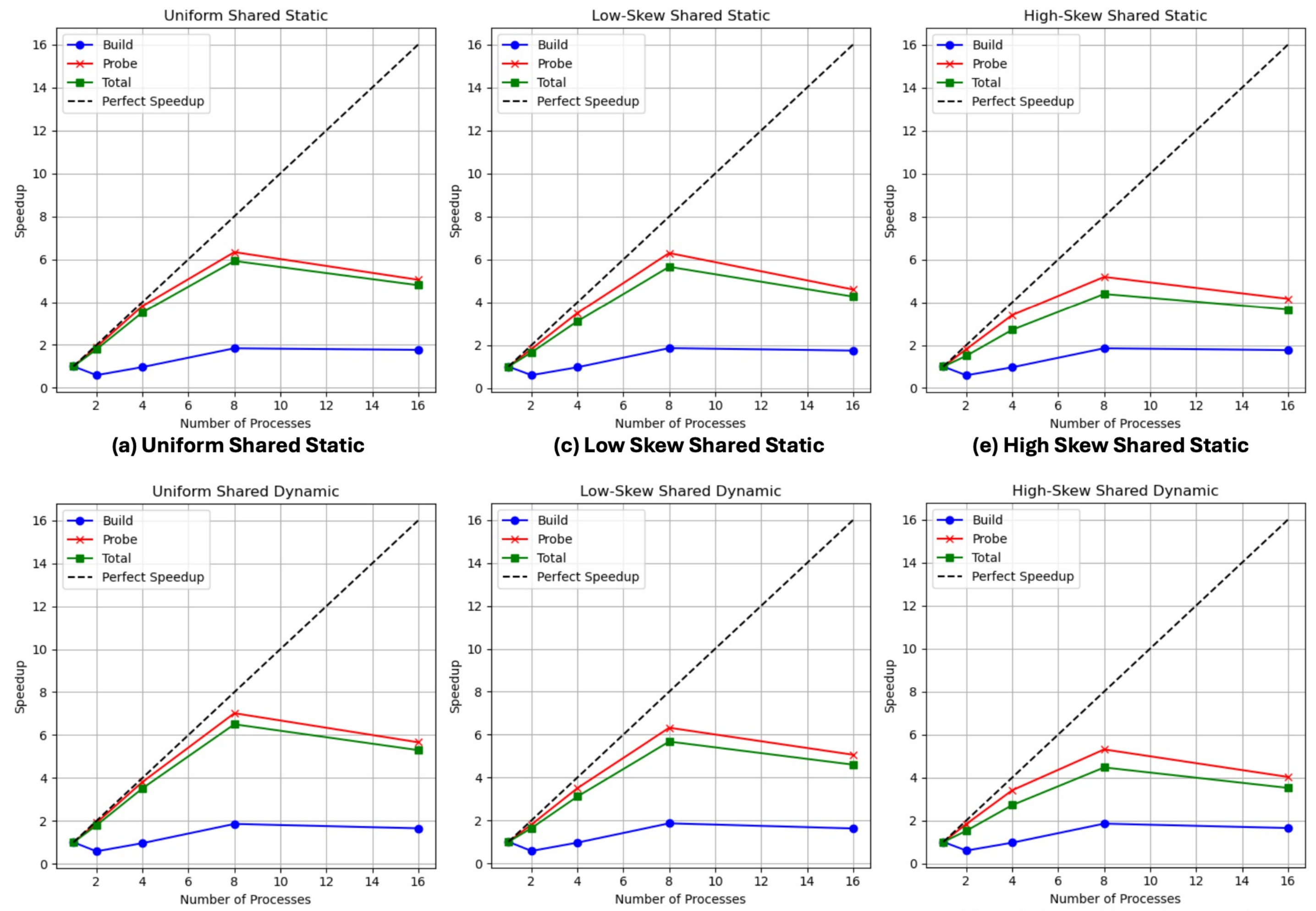
Table 1: Default Parameters Explanation

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

Table 2: Hardware Specification

Speedup of Parallelism

- Parallel hash joins achieve optimal **speedup with increasing thread counts, peaking at 7X speedup for the probe phase when utilizing 8 threads** under uniform data distribution and dynamic task scheduling. **Speedup benefits diminish beyond the maximal hardware parallelism** due to overhead from context switching.
- The **probe phase demonstrates higher speed efficiency due to its read-only data access pattern** requiring no synchronization, contrasting with the build phase that faces significant synchronization overhead from frequent locking and unlocking of hash buckets. **Dynamic scheduling enhances speedup** across all tested workloads by efficiently handling imbalanced workloads.



(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

Cache Analysis

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%**

- Cache miss rates are notably lower in partitioned hash joins across all configurations, supporting the design principle that **smaller, well-sized partitions improve cache efficiency,** particularly evident in uniform workload scenarios.
- Shared hash joins benefit from increased data skew, displaying a significant decrease in cache miss rates as data becomes more skewed, which **enhances temporal cache locality due to the frequent occurrence of hot values.**

Synchronization Overhead Analysis

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

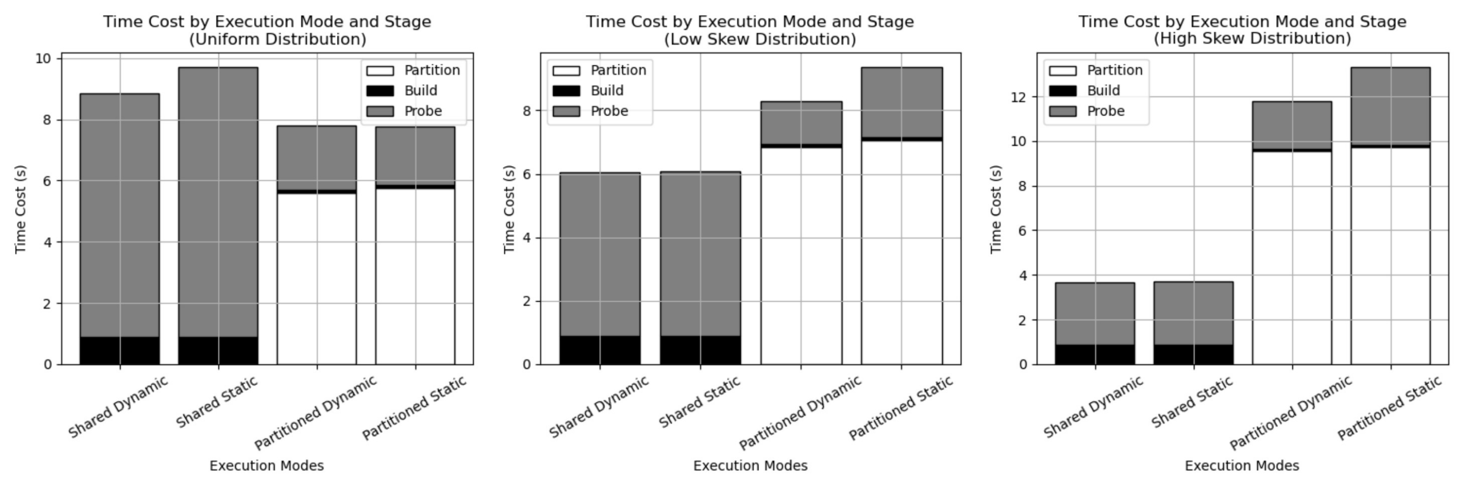
- **Partitioned hash joins generally performing** better in uniform distributions by minimizing thread contention.
- Under low and high **skew distributions, shared hash joins exhibit lower synchronization overhead** compared to partitioned hash joins. This efficiency is due to better workload distribution among threads and reduced contention in highly skewed scenarios.
- Dynamic scheduling strategies typically incur higher CPU cycles due to task reassignment overhead but offer **benefits in balancing workloads across threads, suggesting their preferable use** in varying workload conditions.

Conclusion

- **Shared hash joins excel under skewed data distributions** due to balanced workloads, improved cache locality.
- **Partitioned hash joins are optimal for uniform data distributions** with high cache efficiency.
- Effective database systems should **dynamically select hash join strategies based on workload** to optimize performance.
- High-quality **dynamic task scheduling is essential to address workload imbalances** in partitioned hash joins.
- **Partition number is crucial**. Too few partitions increase overhead and cache misses, while too many amplify cold misses.
- The probe phase (more computationally intensive), requires **optimized hash bucket structures** for better read performance.

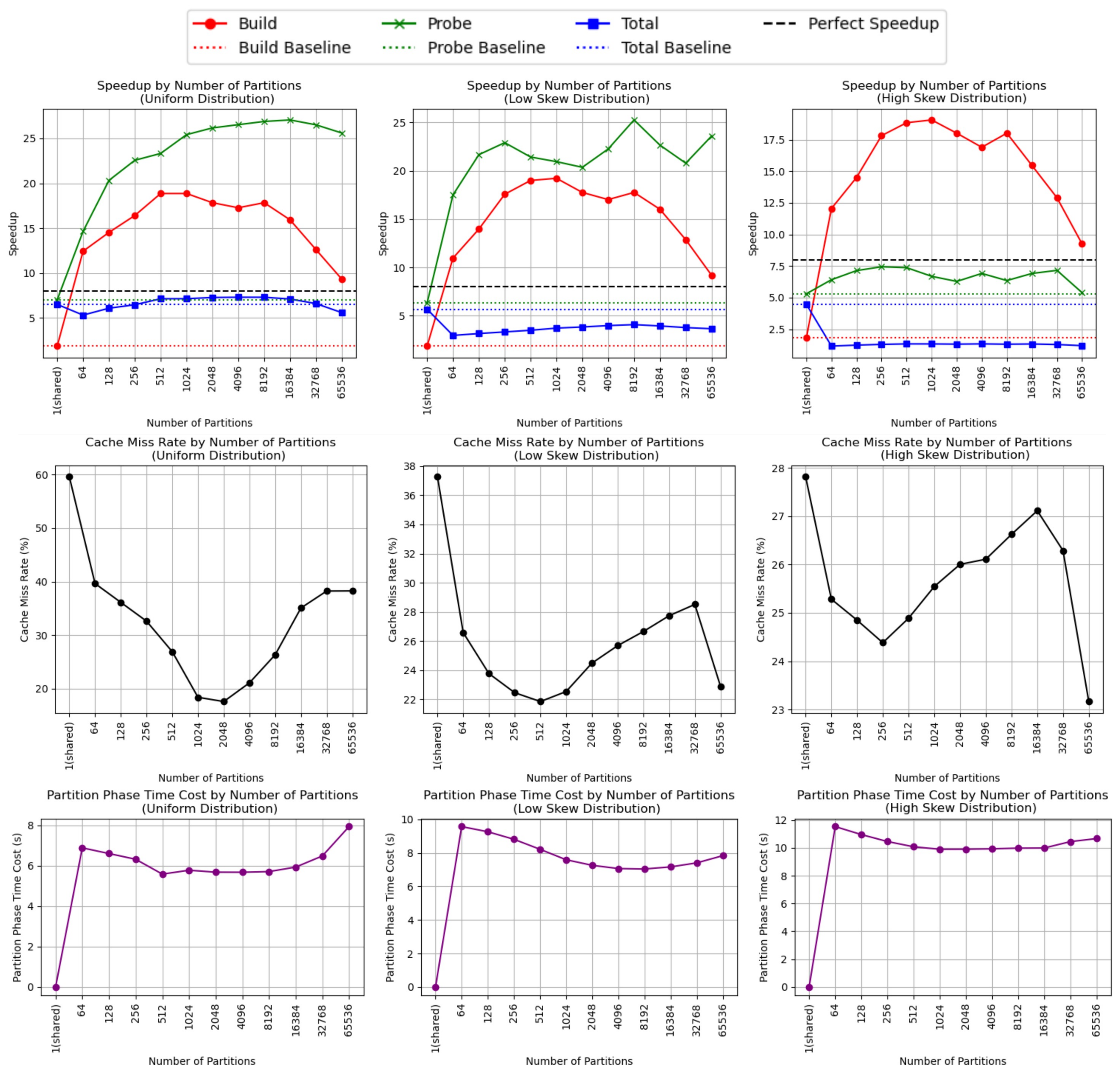
Comparison of Parallelism Patterns

- Performance comparison between parallelism patterns reveals that **partitioned hash joins are optimal for uniform workloads** due to lower cache miss rates, whereas **shared hash joins are more effective under skewed workloads** due to better cache locality and less joins, sensitivity to data distribution.
- **Dynamic task scheduling significantly enhances the performance** of both shared and partitioned hash particularly under skewed workloads, by better balancing thread workloads and minimizing execution times across all phases of the hash join process.



(a) Uniform Distribution; (b) Low Skew Distribution; (c) High Skew Distribution

Effect of Partition Number on Performance



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

- The choice of **partition number in partitioned hash joins is crucial for maximizing performance**, with optimal numbers reducing the largest partition size to improve cache fit and minimize cache capacity miss rates.
- Empirical analysis demonstrates a **non-linear relationship between partition number and performance**: increasing partitions initially improves performance due to reduced cache miss rates, but excessive partitioning leads to increased cold misses as each partition holds fewer tuples.
- Optimal partition numbers **balance the reduction in capacity misses with the rise in cold misses.** For instance, the build phase speedup peaks at partition numbers like 1024 and 2048, highlighting the need to tailor partition numbers to specific workload characteristics to optimize hash join performance.

REFERENCES

[1] Blanas, Spyros, Yinan Li, and Jignesh M. Patel. "Design and evaluation of main memory hash join algorithms for multi-core CPUs." Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. 2011.

[2] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In VLDB, pages 54–65, 1999.

[3] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In DaMoN, pages 25–34, 2008.