# NOCAP: Near-Optimal Correlation-Aware Partitioning Joins


## ABSTRACT

Storage-based joins are still commonly used today because the memory budget does not always scale with the data size. One of the many join algorithms developed that has been widely deployed and proven to be efficient is the Hybrid Hash Join (HHJ), which is designed to exploit any available memory to maximize the data that is joined directly in memory. However, HHJ cannot fully exploit detailed knowledge of the join attribute correlation distribution.

In this paper, we show that given a correlation skew in the join attributes, HHJ partitions data in a suboptimal way. To do that, we derive the optimal partitioning using a new cost-based analysis of partitioning-based joins that is tailored for primary key - foreign key (PK-FK) joins, one of the most common join types. This optimal partitioning strategy has a high memory cost, thus, we further derive an approximate algorithm that has tunable memory cost and leads to near-optimal results, compared to the state-of-the-art. Our algorithm, termed NOCAP (Near-Optimal Correlation-Aware Partitioning) join, outperforms the state-of-the-art for skewed correlations by up to 30%, and the textbook Grace Hash Join by up to 4×. Further, for a limited memory budget, NOCAP outperforms HHJ by up to 10%, even for uniform correlation. Overall, NOCAP dominates previous state-of-the-art algorithms and mimics the best algorithm for a memory budget varying from below the $\sqrt{\|\text{relation}\|}$ to more than $\|\text{relation}\|$.


## 1 INTRODUCTION

Joins are ubiquitous in database management systems (DBMS). Further, *primary key - foreign key* (PK-FK) equi-joins are the most common type of joins. For example, the queries of industry-grade benchmarks like TPC-H [47] and the Join Order Benchmark (JOB)[27] are all PK-FK equi-joins. Recent research has focused on techniques to optimize in-memory equi-joins [4–6, 9, 10, 30, 44, 49], however, as the memory prices scale slower than storage [32], the available memory might not always be sufficient to store both tables simultaneously, thus requiring a classical storage-based join [42]. This is common in a shared resource setting, like multiple colocated databases or virtual database instances that are deployed on the same physical cloud-resident server [8, 12, 23]. Further, in edge computing, memory is also limited, which is further exacerbated when other memory-demanding services are running [20]. In several other data-intensive use cases like Internet-of-things, 5G communications, and autonomous vehicles [14, 17], memory might also be constrained. Finally, there are two main reasons a workload might need to use storage-based joins: (1) workloads consisting of queries with multiple joins, and (2) workloads with a high number of concurrent queries. In both cases, the available memory and computing resources must be shared among all concurrently executed join operators.

**Storage-based Joins.** When executing a join, storage-based join algorithms are used when the available memory is not enough to hold the hash table for the smaller relation. Traditional storage-based join algorithms include Nested Block Join (NBJ), Sort Merge Join (SMJ), Grace Hash Join (GHJ), Simple Hash Join (SMJ), and Hybrid Hash Join (HHJ). Overall, HHJ, which acts as a blend of Simple Hash Join and Grace Hash Join, is considered the state-of-the-art approach [15, 26, 45], and is extensively used in existing database engines (e.g., MySQL [34], PostgreSQL [41], AsterixDB [2]). HHJ **uniformly** distributes records from the two input relations to a number of partitions via hashing the join key and ensures that, when possible, one or more partitions remain in memory and are joined on the fly without writing them to disk. The remaining disk-resident partitions are joined using a classical hash join approach to produce the final result. Unlike GHJ and SMJ, HHJ uses the available memory budget judiciously and thus achieves a lower I/O cost. Existing relational database engines (e.g., PostgreSQL [41] and AsterixDB [2, 24]) often implement a variant of HHJ, Dynamic Hybrid Hash join (DHH) that dynamically decides which partitions should stay in memory during partitioning.

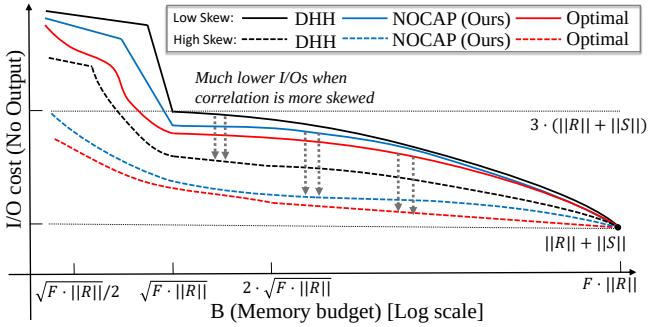



**Figure 1: State-of-the-art dynamical hybrid hash join (DHH) cannot fully exploit the correlation and thus results in much more I/Os than the optimal one when the correlation becomes skewed. Our approximate algorithm uses the same input from DHH and achieves near-optimal I/O cost ($\|R\|$ is the size of the smaller relation in pages, $\|S\|$ is the size of the larger relation, and $F$ is the hash table fudge factor).**

**The Challenge: Exploit Skew.** Although there are many skew optimization for joins [16, 21, 25, 29, 35], a potential detailed knowledge of the join attribute correlation and its skew is not fully exploited since existing techniques use heuristic rules to design the caching and partitioning strategy. In turn, while these heuristics may work well in some scenarios, in general, they lead to suboptimal I/O cost under an arbitrary join correlation. For example, HHJ can be optimized by prioritizing entries with high-frequency keys when building the in-memory hash table [39]. However, practical deployments use limited information about the join attribute correlation and typically employ a fixed threshold for building an in-memory hash table for keys with high skew (e.g., 2% of available memory). As a result, prior work does not systematically quantify the benefit of such approaches, nor does it offer a detailed analysis of how close to optimal they might be. In fact, due to the exponential search space, there is no previous literature that accurately reveals the relationship between the optimal I/O cost and the join correlation, thus leaving a large unexplored space for studying the caching and partitioning strategy.

**Key Idea: Optimal Correlation-Aware Partitioning.** To study the optimality of different partitioning-based join algorithms, we model a general partitioning strategy that allows for arbitrary partitioning schemes (not necessarily based on a specific hash function). We assume that the relations we join have a PK-FK relationship, and we develop an algorithm for *Optimal Correlation-Aware Partitioning* (OCAP) that allows us to compare any partitioning strategy with the optimal partitioning, given the join correlation. Our analysis reveals that the state of the art is suboptimal in the entire spectrum of available memory budget (varied from $\sqrt{\|\text{relation}\|}$ to $\|\text{relation}\|$), leading up to 60% more I/Os than strictly needed as shown in Figure 1, where the black lines are the state of the art, and the red lines are the optimal number of I/Os). The OCAP is constructed by modeling the PK-FK join cost as an optimization problem and then proving the *consecutive theorem* which establishes the basis of finding the optimal cost within polynomial time complexity. We propose a dynamic programming algorithm that finds the optimal solution in quadratic time complexity, $O(n^2 \cdot m^2)$, and a set of pruning techniques that further reduce this cost to $O\left((n^2 \cdot \log m)/m\right)$ (where $n$ is the number of records of the smaller relation and $m$ is the memory budget in pages). We note that OCAP has a large memory footprint as it assumes that the detailed information of the join attributes correlation is readily available and, thus, can be only applied for offline analysis. However, as discussed above, we use OCAP to identify the headroom for improvement from the state of the art.

**The Solution: Near-Optimal Correlation-Aware Partitioning Join.** In order to build a practical join algorithm with a tunable memory budget, we approximate the optimal partitioning provided by OCAP with our *Near-Optimal Correlation-Aware Partitioning* Join (NOCAP) algorithm. NOCAP enforces a strict memory budget and splits the available memory between buffering partitions and caching information regarding keys with a high skew in join correlation. The blue line in Figure 1 shows the I/O cost of the NOCAP join algorithm, which approximates the optimal (red lines) and outperforms the state of the art (DHH, black lines) for any memory budget. Further note that there are two sets of lines: solid lines for low skew and dashed lines for high skew. A key observation is that while DHH is able to exploit higher skew to further reduce its I/O cost, the headroom for improvement for high skew is indeed higher than for low skew, which is largely achieved by our approach. Overall, NOCAP is a practical join algorithm that is mostly beneficial compared to the state of the art, and offers its maximum benefit for a high skew in the join attribute correlation.

**Contributions.** In summary, our contributions are as follows:

- We build a new cost model for partitioning-based PK-FK join algorithms employing arbitrary partitioning schemes.
- We propose an optimal correlation-aware partitioning algorithm (OCAP) based on dynamic programming (§3.1).
- We show that OCAP obtains the optimal I/O cost for offline analysis assuming that the complete join attribute correlation information is given, and its complexity can be reduced to $O\left(n^2 \cdot \log m/m\right)$, while the search space can be as large as $m^n$, where $n$ is the input size and $m$ is the memory budget in pages (§3.2).
- We design an approximate correlation-aware partitioning (NOCAP) join algorithm based on the insights from the optimal partition scheme. NOCAP uses partial correlation information (the same information that is used by state of the art skew optimized join algorithms) and achieves near-optimal performance while respecting memory budget constraints (§4).
- We thoroughly examine the performance of our algorithm by comparing it against GHJ, SMJ, and DHH. We identify that the headroom for improvement is much higher for skewed join correlations by comparing the I/O cost of DHH against the optimal. Further, we show that NOCAP can reach near-optimal I/O cost and thus lower latency under different correlation skew and memory budget, compared to the state of the art (§5). Overall, NOCAP dominates the state-of-the-art and offers performance benefits of up to 30% when compared against DHH and up to 4× when compared against the textbook GHJ.

## 2 PREVIOUS STORAGE-BASED JOINS

We first review four classical storage-based join methods and summarize their costs in Table 1. We also present more details of Dynamic Hybrid Hash Join (DHH), a variant of state-of-the-art HHJ. More detailed descriptions of these algorithms can be found in textbooks and seminal data management papers [15, 19, 22, 42]

### 2.1 Classical Storage-based Joins

**Nested Block Join (NBJ).** NBJ partially loads the smaller relation $R$ (in **chunks** equal to the available memory) in the form of an in-memory hash table and then scans the larger relation $S$ once *per chunk* to produce the join output for the partial data. This process is repeated multiple times for the smaller relation until the entire relation is scanned. As such, the larger relation is scanned for as many times as the number of chunks in the smaller relation.

**Sort Merge Join (SMJ).** SMJ works by sorting both input tables by the join attribute using external sorting and applying $M$-way ($M \leq B-1$) merge sort to produce the join result. During the external sorting process, if the number of total runs is less than $B-1$, the last sort phase can be combined together with the multi-way join phase to avoid repetitive reads and writes [42].

**Grace Hash Join (GHJ).** GHJ uniformly distributes records from the two input relations to a number of partitions (at most $B-1$) via hashing the join key, and the corresponding partitions are joined then. Specifically, when the smaller partition fits in memory, we simply store it in memory (typically as a hash table) and then scan the larger partition to produce the output .

**Hybrid Hash Join (HHJ).** HHJ is a variant of GHJ that allows one or more partitions to stay in memory without being spilled to disk, if the space is sufficient. When partitioning the second relation, we can directly probe in-memory partitions and generate the join output for those in-memory partitions. The remaining keys of the second relation are partitioned to disk, and we execute the same probing phase as in GHJ to join the on-disk partitions. When the memory budget is lower than $\sqrt{\|R\| \cdot F}$ (where $\|R\|$ is the size of the smaller relation in pages, and $F$ is the fudge factor for the in-memory hash table), HHJ downgrades into GHJ because it will not be feasible to maintain a partition in memory while having enough buffers for the remaining partitions and the output.

**Table 1: Estimated Cost for NBJ, GHJ, and SMJ on computing $R \bowtie S$. $\|R\|$ and $\|S\|$ are the number of pages of $R$ and $S$. #chunks is the number of passes for scanning $S$. #pa-runs is the number of times to partition $R$ and $S$ until the smaller partition fits in memory. HHJ$(R,S)$ can be treated as a special case of GHJ$(R,S)$, by replacing #pa-runs with the proportion of $\|R\|$ ($\|S\|$) that is spilled out to disk during partitioning. #s-passes is the number of partially sorted passes of $R$ and $S$, until the number of the total sorted runs is less than $B-1$ ($B$ is the total number of memory available). $\mu$ and $\tau$ indicate the write/read asymmetry, where $RW$, $SW$, and $SR$ denote the latency per I/O for random write, sequential write and sequential read respectively. For simplicity, we assume $\|R\| \leq \|S\|$.**

| Approach | Normalized #I/O | Notation |
|---|---|---|
| NBJ$(R,S)$ | $\|R\| + \#\text{chunks} \cdot \|S\|$ | None |
| GHJ$(R,S)$ | $(1 + \#\text{pa-runs} \cdot (1+\mu)) \cdot (\|R\| + \|S\|)$ | $\mu \stackrel{\text{def}}{=} RW/SR$ |
| SMJ$(R,S)$ | $(1 + \#\text{s-passes} \cdot (1+\tau)) \cdot (\|R\| + \|S\|)$ | $\tau \stackrel{\text{def}}{=} SW/SR$ |

## 2.2 Dynamic Hybrid Hash Join

In this section, we discuss DHH, the state-of-the-art implementation of Hybrid Hash Join, which is also the starting point of our correlation-aware partitioning algorithm. We also discuss the existing skew optimization in DHH.

While the exact implementation of different variants of DHH may differ [22], the core idea is the same across all of them: how to decide which partitions to stay in memory or on disk. We visualize one DHH implementation (AsterixDB [2, 24]) in Figure 2. Compared to HHJ, DHH exhibits higher robustness by dynamically deciding which partitions should be spilled to disk. Specifically, every partition is initially staged in memory, and when needed, a staged partition will be selected to be written to disk, and it will free up its pages for new incoming records. Typically, the largest partition will be selected, however, the victim selection policy may vary in different systems. A bit vector ("Page Out Bits" in Figure 2) is maintained to record which partitions have been written out at the end of the partitioning phase of $R$. Another hash function $h_{probe}$ is applied to build a large in-memory hash table using all staged partitions. When partitioning the relation $S$, we apply the partitioning hash function first and check the page-out bit for every record. If it is set, we directly probe the in-memory hash table using the probe hash function and generate the join result. Otherwise, we move the joined record to the corresponding output page and flush it to the disk if it is full. Finally, we perform exactly the same steps as the traditional hash join to probe all the disk-resident partitions.

**Number of Partitions.** The number of partitions in DHH (noted by $m_{\text{DHH}}$) is a key parameter that largely determines join performance. A large $m_{\text{DHH}}$ requires more output buffer pages reserved for each partition, and, thus, less memory is left for memory-resident partitions. On the other hand, small $m_{\text{DHH}}$ renders the size of each partition very large, which makes it harder to stage in memory. No previous work has formally investigated the optimal number of partitions for arbitrary correlation skew, but past work on HHJ employs some heuristic rules. For example, an experimental study [22] recommends that 20 is the minimum number of partitions when

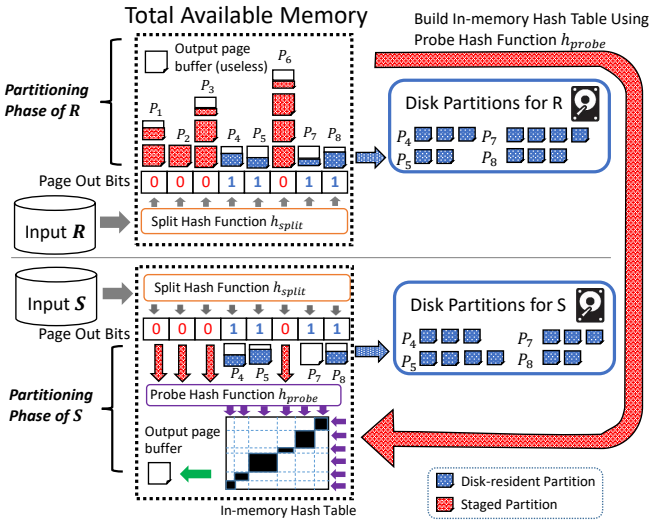



**Figure 2: In the above Hybrid Hash Join, $P_1, P_2, P_3, P_6$ (marked by red) are staged in memory when the input relation $S$ is being partitioned, while the remaining partitions (marked by blue) are written to disk.**

we do not have sufficient information about the join correlation. In addition, if we restrict the number of memory-resident partitions to be 1, we can set the number of partitions $m_{\text{DHH}}$ as follows [45]:

$$m_{\text{DHH}} = \left\lceil \frac{\|R\| \cdot F - B}{B - 1} \right\rceil \quad (1)$$

where $\|R\|$ is the size of smaller partition in pages, $F$ is the fudge factor for the hash table, and $B$ is the total given memory budget in pages. Integrating this equation with the above tuning guideline, we configure $m_{\text{DHH}} = \max\{\text{Equation (1)}, 20\}$.

**Heuristic Skew Optimization In Practical Deployments.** For ease of notation, we assume that $R$ is the relation that holds the primary key and is smaller in size (dimension table) and that $S$ is the relation that holds the foreign key and is larger in size (fact table). In PK-FK joins the distribution of the join key in the fact table can be non-uniform. Thus, caching entries with high-frequency keys can substantially reduce the I/O cost. Specifically, if there is enough memory budget, we first partition the dimension table $R$ and cache entries with high-frequency keys in a small hash table in memory. When we partition the fact table $S$, if there are any keys matched in the small hash table, we directly join them and output. That way, we avoid writing out many entries with the same key from the fact table and reading them back into memory during the probing phase. However, it is unclear which frequency should be treated as *high* to qualify for this optimization and how large the hash table for the frequent keys should be within the available memory budget. In the state-of-the-art implementation from PostgreSQL [39], keys with high frequency in the sampled data will be collected and stored in a system cache. When the skew optimization knob is on, PostgreSQL will try to build the in-memory hash table for skewed keys with a fixed budget of 2% memory. Then it sums up the frequency of keys in the hash table, and if the summed frequency is no more than 1% of the whole input size of relation $S$, the skew optimization will not be triggered as the skew is deemed insufficient. Existing skew optimizations, which are based on fixed thresholds, reduce

the number of I/Os for skewed join correlation. However, as shown in Figure 1, they are not enough to achieve ideal I/O reduction, and they are further away from the optimal for higher skew.

# 3 OPTIMAL CORRELATION-AWARE PARTITIONING JOIN

In this section, we discuss the theoretical limit of an I/O-optimal partitioning-based join algorithm when the correlation skewness information is known in advance and can be accessed for free. More specifically, we model the correlation between two input relations with respect to the join attributes as a *correlation table* CT, such that $\mathsf{CT}[i]$ represents the number of matching records in the outer relation $S$ for the $i$-th record in the inner relation $R$. For simplicity, we assume that $R$'s join attribute is the primary key, and $S$'s join attribute is the foreign key. In addition, we assume that $R$ is the smaller relation that is used to build the in-memory hash table (the case when $S$ is the smaller relation can be tackled similarly with slight difference in the cost function). Our goal is to find which keys should be cached in memory, and how the rest keys are partitioned on disk that would minimize the total I/Os in the join execution under arbitrary memory budget.

To find such an optimal partitioning, we start with an easy case when no records can be cached during the partition phase (§3.1). After dealing with this case, we next turn to the general case when keys can be cached (§3.2). We note again that the memory used for storing CT as well as the optimal partitioning does not compete with the available memory budget $B$, which is indeed **unrealistic**. This is why we consider it as a theoretically I/O-optimal algorithm, since it can help us understand the lower bound of any feasible partitioning but cannot be converted into a practical algorithm.

## 3.1 Optimal Partitioning Without Caching

In this section, we build the cost model which assumes that there are no cached records during the partitioning phase. We formulate this problem as an integer program (§3.1.1), present the *consecutive theorem* (§3.1.2) and show an optimal solution via dynamic programming (§3.1.3).

### 3.1.1 Partitioning as An Integer Program.

Any partitioning is essentially an assignment of $n$ records from the inner relation $R$ into $m$ partitions, where $m \leq B-1$ and we assume it is given for now (it will be decided in Section 3.2 and Section 4.2). As we focus on PK-FK joins, once we know how to partition $R$ (the relation with the primary key), we can apply the same partition strategy to $S$.

*Partitioning.* We encode a partitioning strategy into a Boolean matrix $\mathbb{P}$ of size $n \times m$, such that $\mathbb{P}_{i,j} = 1$ indicates that the $i$-th record belongs to the $j^{\text{th}}$ partition. If we know some records do not have any matching record from $S$ in advance, i.e., the corresponding CT values is 0, we will not assign it to any partition. We can pre-process the records to filter out these records so that each of the remaining records belong to exactly one partition. Hence, we have $\sum_{j=1}^{m} \mathbb{P}_{i,j} = 1$ for every $i \in [n]$. We note that in reality it is impossible to remove records that do not have any matching records during the partitioning phase. In the partitioning phase, we use at least one page to stream the input relation and the rest of the pages as output buffers for each partition, hence, $m \leq B-1$. Using the notation of the partition matrix, the size (in pages) of a partition $R_j$ is given by $\|R_j\| = \lceil \sum_{i=1}^{n} \mathbb{P}_{i,j} / b_R \rceil$, and the size of the corresponding partition $S_j$ is given by $\|S_j\| = \lceil (\sum_{i=1}^{n} \mathbb{P}_{i,j} \cdot \mathsf{CT}[i]) / b_S \rceil$, where $b_R$ (resp. $b_S$) is the number of records per page for $R$ (resp. $S$).

*Cost Function.* It remains to join partitions in a pair-wise manner. We consider a light optimizer that picks the most efficient join algorithm among NBJ, GHJ and SMJ. We reuse the cost model in Table 1 and pick the algorithm that achieves the minimum I/O cost. Therefore, the *per-partition join cost* is modeled as:

$$\mathsf{PPJ}(R_j, S_j) = \min\{\mathsf{NBJ}(R_j, S_j), \mathsf{SMJ}(R_j, S_j), \mathsf{GHJ}(R_j, S_j)\}$$

We omit DHH here for simplicity, which does not lead to major changes of the optimal partitioning, since in most cases NBJ is the most efficient one when taking read/write asymmetry into consideration (write is generally much slower than read in modern SSDs [36, 37]). The *join cost* induced by partition $\mathbb{P}$ is $\mathsf{Join}(\mathbb{P}) = \sum_{j=1}^{m} \mathsf{PPJ}(R_j, S_j)$. For ease of illustration, from now on, we assume only NBJ is applied in the partition-wise join. It can be proved that most of the following lemmas, theorems, corollaries still apply when NBJ, SMJ, and GHJ are all used in the partition-wise joins as long as we always pick the one with minimum cost (detailed proof is omitted due to limited space). Recall the cost function of NBJ in Table 1. By setting #chunks= $\lceil \|R_j\| / ((B-2)/F) \rceil$, we come to objective function as follows:

$$\begin{aligned}
\mathsf{Join}(\mathbb{P}) &= \sum_{j=1}^{m} \left( \|R_j\| + \left\lceil \frac{\|R_j\|}{(B-2)/F} \right\rceil \cdot \|S_j\| \right) \\
&= \sum_{j=1}^{m} \left( \left\lceil \sum_{i=1}^{n} \mathbb{P}_{i,j} / c_R \right\rceil \cdot \left( \sum_{i=1}^{n} \mathbb{P}_{i,j} \cdot \mathsf{CT}[i] \right) \right) + \|R\| \\
&= \sum_{i=1}^{n} \mathsf{CT}[i] \cdot \left( \sum_{j=1}^{m} \mathbb{P}_{i,j} \cdot \left\lceil \sum_{l=1}^{n} \mathbb{P}_{l,j} / c_R \right\rceil \right) + \|R\|
\end{aligned}$$

where $c_R = \lfloor b_R \cdot (B-2)/F \rfloor$ is a constant that denotes the number of records per chunk in relation $R$. Note that the I/O cost in the first partition phase is fixed for all partition strategies, since any partition scheme has to read and write both relations. And we can also assume that $\sum_{j=1}^{m} \|R_j\| = \|R\|$ is a fixed, which does not depend on how we partition records, and thus we can further omit $\|R\|$ in the above objective function. Finding the partitioning with minimum join cost can be captured by the following integer program:

$$\begin{aligned}
\min_{\mathbb{P}} \quad & \mathsf{Join}(\mathbb{P}) \\
\text{subject to} \quad & \sum_{j=1}^{m} \mathbb{P}_{i,j} = 1, \forall i \in [n] \\
& \mathbb{P}_{i,j} \in \{0,1\}, \forall i \in [n], j \in [m]
\end{aligned}$$

### 3.1.2 Consecutive Theorem.

As each record belongs to exactly one partition, we use a function $f$ to denote the mapping from $i$ to the index of partition $f(i)$ where the $i$-th record is assigned. Moreover, we use $\mathcal{N}(i) = \{i^* \in [n] : f(i^*) = f(i)\}$ to denote the records that fall into the same partition

with the $i$-th record. This way, we can rewrite the join cost as:

$$\text{Join}(\mathbb{P}) = \sum_{i=1}^{n} \text{CT}[i] \cdot \left\lceil \sum_{l=1}^{n} \mathbb{P}_{l,f(i)}/c_R \right\rceil = \sum_{i=1}^{n} \text{CT}[i] \cdot \left\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \right\rceil \quad (2)$$

Observe from Equation (2) that the overall join cost can be also calculated by aggregating $\text{CT}[i] \cdot \lceil \|\mathcal{N}(i)\|/c_R \rceil$ for each record. For ease of analysis, we assume that CT is *sorted in ascending order*, i.e., $\text{CT}[i_1] \le \text{CT}[i_2]$ for any $1 \le i_1 \le i_2 \le n$. We notice that when CT is already sorted, some nice property of (2) can be further exploited, which leads to Lemma 3.1.

Lemma 3.1. *For any partitioning $\mathbb{P}$, if there exists a pair of indexes $i_1, i_2$ such that $CT[i_1] > CT[i_2]$ and $\|\mathcal{N}(i_1) \ge \|\mathcal{N}(i_2)\|$, it is always feasible to find a new partition $\mathbb{P}'$ by swapping records at $i_1$ and $i_2$, which always achieves lower (or at least not higher) cost than $\mathbb{P}$.*

Proof. After swapping records at $i_1$ and $i_2$, all remaining records their aggregated value $\text{CT}[i] \cdot \lceil \|\mathcal{N}(i)\|/c_R \rceil$ preserved, since $\mathcal{N}(i)$ stays the same for every $i \in [n] - \{i_1, i_2\}$. As such, we have:

$$\begin{aligned}
&\text{Join}(\mathbb{P}') - \text{Join}(\mathbb{P}) \\
&= \text{CT}[i_1] \cdot \lceil \|\mathcal{N}(i_2)\|/c_R \rceil + \text{CT}[i_2] \cdot \lceil \|\mathcal{N}(i_1)\|/c_R \rceil \\
&\quad - \text{CT}[i_1] \cdot \lceil \|\mathcal{N}(i_1)\|/c_R \rceil - \text{CT}[i_2] \cdot \lceil \|\mathcal{N}(i_2)\|c_R \rceil \\
&= (\text{CT}[i_1] - \text{CT}[i_2]) \cdot (\lceil \|\mathcal{N}(i_2)\|/c_R \rceil - \lceil \|\mathcal{N}(i_1)\|/c_R \rceil) \le 0
\end{aligned}$$

hence $\text{Join}(\mathbb{P}') \le \text{Join}(\mathbb{P})$ holds. □

With Lemma 3.1, we come to the core property of the optimal partitioning with respect to the sorted CT:

Theorem 3.2 (Consecutive Theorem). *There exists an optimal partitioning $\mathbb{P}_{opt}$ where each partition contains "consecutive" records from the sorted CT. Formally, there exists a mapping function $f_{opt} : [n] \to [m]$, such that for any $1 \le i_1 \le i_2 \le n$, if $f_{opt}(i_1) = f_{opt}(i_2)$, then $f_{opt}(j) = f_{opt}(i_1)$ holds for any $j \in [i_1, i_2]$.*

Proof. We show that it is always feasible to convert any partitioning $\mathbb{P}$ (including an arbitrary optimal partitioning $\mathbb{P}_{opt}$) into $\mathbb{P}'$ in which each partition contains consecutive records from sorted CT, while $\text{Join}(\mathbb{P}') \le \text{Join}(\mathbb{P})$. Now, we consider an arbitrary partitioning $\mathbb{P}$. Let $P_j$ be the set of records falling into the $j$-th partition, for $1 \le j \le m$. W.l.o.g., we assume that records in $P_j$ lie in a non-consecutive way in sorted CT array. See Figure 3. Let $i_j^-$ and $i_j^+$ be the minimum and maximum index of records from $P_j$ lying on CT. Let $S_j^1, S_j^2, \cdots, S_j^\ell$ be the longest consecutive records whose indexes fall into $[i_j^-, i_j^+]$ in CT, and belong to the same partition $P_{j'}$ for some $j' \ne j$ ($S_j^{\ell_1}$ and $S_j^{\ell_2}$ may belong to different partitions). We denote $S_j = S_j^1 \cup S_j^2 \cup \cdots \cup S_j^\ell$ as the set of all records that make $P_j$ non-consecutive with respect to the sorted CT.

The $\mathbb{P}'$ can be obtained from $\mathbb{P}$ via a series of swap operations, as described in Algorithm 1. In this process, we always identify a partition whose records are not consecutive on the sorted CT, say $P_j$. We also maintain two pointers left and right to indicate the smallest and largest indexes of records to be swapped. Consider the record at the smallest index $i \in S_j$. We swap it with the record at left or right in CT. More specifically, if $\|\mathcal{N}(i)\| > \|P_j\|$, we swap $i$ and left by putting the record at $i$ into partition $P_{f(\text{left})}$ and putting the record at left into partition $P_{f(i)}$ (By definition, we have $P_{f(i)} = \mathcal{N}(i)$). Otherwise, we swap $i$ and right by putting the record at $i$ into partition $P_{f(\text{right})}$ and putting the record at right into partition $P_{f(i)}$.

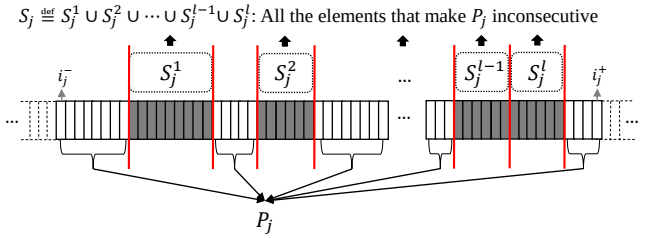



**Figure 3: An illustration of records from one partition $P_j$ lying in a non-consecutive way in the sorted CT.**

**Algorithm 1:** SWAP($\mathbb{P}$)

```
1  while there exists P_j ∈ ℙ that is non-consecutive do
2      while True do
3          left ← min_{i∈[n]:f(i)=j} i;
4          right ← max_{i∈[n]:f(i)=j} i;
5          S_j ← {i ∈ [n] : left ≤ i ≤ right, f(i) ≠ j};
6          if S_j == ∅ then break;
7          ℓ ← min_{i∈S_j} i;
8          if ‖𝒩(ℓ)‖ > ‖P_j‖ then
9              swap(f(ℓ), f(left));
10             left ← left + 1;
11         else
12             swap(f(ℓ), f(right));
13             right ← right − 1;
14 return ℙ;
```

We first show that the join cost does not increase in this swap process. Once a swap operation is invoked at line 9, we always have $\text{left} \le \ell$ and $\|P_j\| = \|\mathcal{N}(\text{left})\| \le \|\mathcal{N}(\ell)\|$, hence Lemma 3.1 indicates that this swap does not increase the join cost. Similarly, when a swap operation is invoked at line 12, we always have $\ell \le \text{right}$ and $\|\mathcal{N}(\ell)\| \le \|\mathcal{N}(\text{right})\| = \|P_j\|$, hence Lemma 3.1 indicates that this swap does not, either, increase the join cost.

Next we show that a consecutive partition will never change to a non-consecutive one, i.e., the swap operations do not harm other consecutive partitions. At first, the size of each partition does not change in the swap process. Consider an arbitrary consecutive partition $P_{j'}$. If it never appears in $S_j$ for some non-consecutive partition $P_j$, then none of its records gets swapped, so it is always consecutive. Otherwise, we assume that $P_{j'}$ appears in $S_j$ as a consecutive sub-array, say $P_{j'} = S_j^k$. If $\|P_{j'}\| \le \|P_j\|$, all records in $P_{j'}$ will always be swapped with the right-most records (although right gets decreased by 1 after each swap), which also form a consecutive sub-array of sorted CT. Symmetrically, if $\|P_{j'}\| > \|P_j\|$, all the records in $P_{j'}$ will always be swapped with the left-most records (although left gets increased by 1 after each swap), which also form a consecutive sub-array of sorted CT. This way, $P_{j'}$ is always consecutive in the swap process. Therefore, as long as we iteratively apply the swapping strategy to every non-consecutive

**Algorithm 2:** PARTITION(CT, $n, m$)

```
1  Initialize V of sizes n × m;
2  for i ← 1 to n do
3      V[i][1].cost ← CalCost(1, i);
4      V[i][1].lastPos ← 1;
5  for i ← 2 to n do
6      for j ← 2 to min(m, i) do
7          V[i][j].cost ← +∞;
8          for k ← 0 to i − 1 do
9              tmp ← V[k][j − 1].cost + CalCost(k + 1, i);
10             if tmp < V[i][j].cost then
11                 V[i][j].cost ← tmp;
12                 V[i][j].lastPos ← k + 1;
13 return V;
```

**Algorithm 3:** GETCUT($V, n, m$)

```
1  lastPos ← n;
2  for j ← 0 to m − 1 do
3      for i ← V[lastPos][m − j].lastPos to lastPos do
4          f(i) ← m − j;
5      lastPos ← V[lastPos][m − j].lastPos − 1;
6  return f;
```

**Algorithm 4:** SORT($\mathbb{P}$)

```
1  while there exists P_{j'} ≾ P_j ∈ ℙ s.t. ‖P_{j'}‖ < ‖P_j‖ do
2      ℓ ← min_{i∈P_j} i;
3      ℓ' ← min_{i∈P_{j'}} i;
4      for i ← 0 to ‖P_{j'}‖ do SWAP(f(ℓ + i), f(ℓ' + i));
5      ℙ ← SWAP(ℙ);
```

partition of $\mathbb{P}$, we will eventually obtain another partitioning $\mathbb{P}'$ where each partition contains consecutive records in CT. □

### 3.1.3 Dynamic Programming.

With Theorem 3.2, we can now reduce the search complexity using dynamic programming. Instead of iterating all possible partitionings, we only need to look for the optimal solution when each partition contains consecutive records in the sorted CT. We first define a sub-problem parameterized by $(i, j)$: find the optimal partitioning for the first $i$ records in CT using $j$ partitions. We can smartly compute $V$ in a bottom-up fashion, and our original problem can be solved by $V[n][m]$. Details of this dynamic programming are described in Algorithm 2. More specifically, $V[i][j]$.cost stores the cost of an optimal partitioning for the sub-problem parameterized by $(i, j)$ and $V[i][j]$.lastPos stores the starting position of the last partition correspondingly. We also define $\mathsf{CalCost}(s, e)$ as the join cost $\mathsf{PPJ}(R_{j'}, S_{j'})$ (omitting the term $\|R_{j'}\|$ in $\mathsf{NBJ}(R_j, S_j)$ for a pair of partition, if we let $P_{j'} = \{i \in [s, e]\}$). This can be easily computed using the pre-computed prefix sum, such that

$$\mathsf{CalCost}(s, e) = (\mathsf{PCT}[e] - \mathsf{PCT}[s-1]) \cdot \lceil \frac{e - s + 1}{c_R} \rceil$$

where $\mathsf{PCT}[k] = \sum_{i=1}^{k} \mathsf{CT}[i]$. The essence of Algorithm 2 is the following recurrence formula:

$$V[i][j].\text{cost} = \min_{0 \le k \le i-1} \{V[k][j-1].\text{cost} + \mathsf{CalCost}(k+1, i)\}$$

which iteratively search the starting position of the last partition in $[0, i-1]$. Also, we can backtrack the index of the last partition to produce the mapping function $f : [n] \to [m]$ from each index to its associated partition, which serves as a partitioning $\mathbb{P}$ for all records. The details are presented in Algorithm 3.

**Complexity Analysis.** Note that CalCost() is computed in $O(1)$ time. Summing over these three for-loops, the total time complexity is $O(m \cdot n^2)$. This is much better than enumerating all possible partitioning strategies, of which the time complexity can reach $O(m^n)$. Regarding the space, storing $V$ results in $O(m \cdot n)$ cost.

### 3.1.4 Pruning.

We further propose two pruning techniques – *monotonicity-based pruning* and *step-based pruning* – to reduce the time and space complexity of the dynamic programming. The core idea is to reduce unnecessary computations of $V$ in Algorithm 2.

*Monotonicity-based Pruning.* We start from the optimal partitioning scheme $\mathbb{P}$ derived by Algorithm 3. As each partition in $\mathbb{P}$ contains consecutive records in the sorted CT, we can define a partial ordering $\precsim$ on the partitions in $\mathbb{P}$. For simplicity, we assume that $P_1 \precsim P_2 \precsim \cdots \precsim P_m$. For a specific partition $P_j$, if there exists a smaller partition $P_{j'}$ (i.e., $\|P_{j'}\| < \|P_j\|$) located ahead of $P_j$ in the sorted CT (i.e., $P_{j'} \precsim P_j$), we can apply Lemma 3.1 to swap records between $P_{j'}$ and $P_j$. This swap would turn $P_j$ into non-consecutive. In this case, we further invoke Algorithm 1 to transform $P_j$ into a consecutive one; moreover, $P_{j'}$ will be put after $P_j$ in the sorted CT. As described in Algorithm 4, the procedure above can be repeatedly applied until $\mathbb{P}$ is *sorted* in terms of the partition size, i.e., $\|P_1\| \ge \|P_2\| \ge \cdots \ge \|P_m\|$. Hence, we come to Theorem 3.3.

THEOREM 3.3 (SORT THEOREM). *There exists an optimal partitioning $\mathbb{P} = (P_1, P_2, \cdots, P_m)$, where each $P_i$ contains consecutive records from sorted CT array, and $\|P_1\| \ge \|P_2\| \ge \cdots \ge \|P_m\|$.*

Combing Theorem 3.3 with the pigeonhole principle, we know that for any sub-problem parameterized by $(i, j)$, there exists an optimal partitioning $\mathbb{P}$ such that: 1) the largest partition contains at least the first $\left\lfloor \frac{i-1}{j} \right\rfloor + 1$ records in sorted CT, and 2) the smallest partition contains at most the last $\left\lfloor \frac{i}{j} \right\rfloor$ records in sorted CT. Hence, we can apply this pruning at line 8 ($k \leftarrow 0$ to $i-1$) in Algorithm 2:

- The partition $P_j$ contains records from $\mathsf{CT}[k+1 : i]$, and $\|P_j\|$ should be no larger than the last (smallest) partition in $V[k][j-1]$. Based on our analysis above, the last (smallest) partition for the sub-problem parameterized by $(k, j-1)$ is at most $\left\lfloor \frac{k}{j-1} \right\rfloor$. We thus have:

$$i - k \le \lfloor \frac{k}{j-1} \rfloor \le \frac{k}{j-1} \Rightarrow k \ge i \cdot (1 - \frac{1}{j}) \tag{3}$$

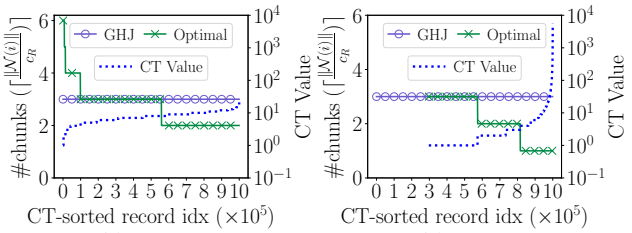



**Figure 4: GHJ randomly partitions and thus the number of required passes ($\lceil \|\mathcal{N}(i)\|/c_R \rceil$) for each partition is the same, regardless of the correlation. In contrast, using our optimal partitioning algorithm, the number of required passes follows a non-monotonically decreasing trend as CT values increase (excluding records with $\mathsf{CT}[i] = 0$).**

- The partition $P_j$ contains records from $\mathsf{CT}[k+1:i]$, and it should be no smaller than the largest partition of remaining records in $\mathsf{CT}[i+1:n]$. Recall that, we still need to put the remaining $n-i$ records into $m-j$ partitions. In the optimal solution of the sub-problem parameterized by $(n-i, m-j)$, the largest partition contains at least $\left\lfloor \frac{n-i-1}{m-j} \right\rfloor + 1$ records. Hence we have,

$$i - k \geq \lfloor \frac{n-i-1}{m-j} \rfloor + 1 \Rightarrow k \leq \max\{i - \lfloor \frac{n-i-1}{m-j} \rfloor - 1, 1\} \quad (4)$$

After changing the range of the third loop to $\lceil \max(i - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1, 1), i \cdot (1 - \frac{1}{j}) \rceil$, we can skip the calculation of $V[i][j]$ when $i \cdot (1 - \frac{1}{j}) < i - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1$. This pruning further reduces the time complexity to $O(n^2 \log m)$.

*Step-based Pruning.* The optimal value of $\mathsf{Join}(\mathbb{P})$ is largely affected by the ceiling function. In an extreme case, $\|P_j\| = 1$ will have the same value $\lceil \|P_j\|/c_R \rceil$ with $\|P_j\| = c_R = \lfloor b_R \cdot (B-2)/F \rfloor$. To utilize this in further pruning, we come to Theorem 3.4:

Theorem 3.4 (Divisible Theorem). *There exists an optimal partitioning $\mathbb{P}$ where the size of all partitions except the largest partition, is divisible by $c_R$.*

The detailed proof of Theorem 3.4 is omitted due to limited space. To utilize this property in Algorithm 2, we initially put the first ($n \bmod c_R$) records in the first partition, and then change the step size (line 5 and line 8) of Algorithm 2 to $c_R$. Alternatively, we can also shrink the size $V$ to $(\lceil n/c_R \rceil + 1) \times m$, based on Theorem 3.4. Recall that $m$ can be as large as $B-1$, $F$ is a constant larger than 1, and thus $c_R = \lfloor b_R \cdot (B-2)/F \rfloor \geq m$. Therefore, the time complexity can be reduced to $O\left(\left(\frac{n}{\lfloor b_R \cdot (B-2)/F \rfloor}\right)^2 \log m\right) = O\left(\frac{n^2 \log m}{m^2}\right)$ and the space complexity now becomes $O\left(\frac{n}{\lfloor b_R \cdot (B-2)/F \rfloor} \cdot m\right) = O(n)$. We now use an example to illustrate the difference between the optimal partitioning and the uniform partitioning.

*Example 3.5.* We consider the case when $B \leq \sqrt{\|R\| \cdot F}$. Note that DHH without skew optimization will downgrade into GHJ when the available buffer is not sufficient to store one partition. Suppose we join relation $R$ with $S$, with $n_R = 1M$, $n_S = 8M$ keys, and the join key is the primary key in $R$ and the foreign key in $S$. Assuming the record size of $R$ is $1KB$, the relation occupies $250K$ pages. We fix $F$ as 1.02 and $B$ as 320 pages in this example.

As shown in Figure 4a, after applying random (uniform) partitioning (GHJ), the partition size is around $250K/319 \approx 784$ pages which require $\lceil 783/(318 \times F) \rceil = 3$ passes to scan every partition from $S$. In contrast, the optimal partitioning allows the number of passes to vary from 2 to 6. Records with low CT values have more passes (e.g., 4,5,6) and records with high CT values have fewer passes (e.g., 2), which is consistent with Theorem 3.3. We can also observe a similar pattern when we have skewed correlation, as shown in Figure 4b (the optimal partitioning naturally excludes records that do not have any match, e.g., $\mathsf{CT}[i] = 0$).

## 3.2 Optimal Partitioning With Caching

We now consider the general case when records can be cached. In this case, some records stay in memory during the partition phase, so as to avoid repetitive reads/writes, while remaining records still go through the partitioning phase as Section 3.1. A natural question arises: which records should stay in memory and which records should go to partition? As we can see in DHH (Figure 2), the hash function $h_{\mathsf{split}}$, together with Page Out Bits, actually categorizes all the records from $R$ into two sets, memory-resident data as $D_{\mathsf{mem}} \subseteq [n]$ and disk-resident data as $D_{\mathsf{disk}} = [n] - D_{\mathsf{mem}}$. Then, it suffices to find an optimal partitioning (without caching) for $D_{\mathsf{disk}}$. Together, we derive a general join cost function $\mathsf{Join}(D_{\mathsf{disk}}, \mathbb{P})$ as:

$$\left\lceil \sum_{i \in D_{\mathsf{disk}}} \frac{\mathsf{CT}[i]}{b_S} \cdot \left\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \right\rceil \right\rceil + (1+\mu) \cdot \left\lceil \frac{\|D_{\mathsf{disk}}\|}{b_R} \right\rceil + \mu \cdot \left\lceil \sum_{i \in D_{\mathsf{disk}}} \frac{\mathsf{CT}[i]}{b_S} \right\rceil$$

where $b_R$ (resp. $b_S$) represents the number of records per page for relation $R$ (resp. $S$), and $\mu$ is the ratio between random write and sequential read. Compared to the original cost function $\mathsf{Join}(\mathbb{P})$, the new cost function $\mathsf{Join}(D_{\mathsf{disk}}, \mathbb{P})$ involves two additional terms – $\mu \cdot (\lceil \|D_{\mathsf{disk}}\|/b_R \rceil + \lceil \sum_{i \in D_{\mathsf{disk}}} \mathsf{CT}[i]/b_S \rceil)$ as the cost of partitioning both relations, and $\lceil \|D_{\mathsf{disk}}\|/b_R \rceil$ as the cost of loading $R$ in the probing phase. Together, we have the following integer program:

$$\begin{aligned} \min_{D_{\mathsf{disk}}, \mathbb{P}} \quad & \mathsf{Join}(D_{\mathsf{disk}}, \mathbb{P}) \\ \text{subject to} \quad & m + \left\lceil \frac{n - \|D_{\mathsf{disk}}\|}{b_R} \cdot F \right\rceil \leq B - 2, \\ & \sum_{j=1}^{m} \mathbb{P}_{i,j} = 1, \forall i \in [n] \end{aligned}$$

where the first constraint limits the size of memory-resident hash table no larger than $B-2-m$, since 2 pages are reserved for input and join output, and $m$ pages are reserved as the output buffer for each disk-resident partition.

So far, the integer program above is parameterized by $D_{\mathsf{disk}}$. It remains to identify the optimal $D_{\mathsf{disk}}$ (i.e., which records should be spilled out to disk) that leads to the overall minimum cost. Although there is a exponentially large number of candidates for $D_{\mathsf{disk}}$, it is easy to see that records with lowest CT value should be spilled out. In other words, Theorem 3.2 still applies for this hybrid partitioning – we can achieve the optimal cost by caching the top-$k$ records from CT in memory, if restricting the size of $D_{\mathsf{mem}}$ to be $k$ ($k < c_R$ since at most $c_R$ records can remain in memory). Thus, by extending

**Algorithm 5:** OCAP(CT, $n$, $B$)

```
tmp ← +∞, f ← ∅;
for k ← 0 to c_R do
    V ← Partition(CT[1 : n − k], n − k, B − 2 − ⌈k·F / b_R⌉);
    c_probe ← ⌈(n−k)/b_R⌉ + V[n − k][B − 2 − ⌈k·F/b_R⌉].cost;
    c_part ← μ · (⌈(n−k)/b_R⌉ + ⌈PCT[n−k]/b_S⌉);
    if tmp < c_probe + c_part then
        tmp ← c_probe + c_part;
        f ← GetCut(V, n − k, B − 2 − ⌈k·F/b_R⌉)
return f;
```

Algorithm 2 to a hybrid one, we finalize the Optimal Correlation-Aware Partitioning algorithm (OCAP), as shown in Algorithm 5.

**Complexity Analysis.** As the number of records that can remain in memory is at most $c_R = \lfloor b_R \cdot (B-2)/F \rfloor$, we run Algorithm 2 at most $c_R$ times. Recall that the complexity of Algorithm 2 can be reduced to $O(\frac{n^2 \log m}{c_R^2})$, the time complexity for OCAP is now $O(\frac{n^2 \log m}{c_R}) = O(\frac{n^2 \log m}{m})$. The space complexity remains $O(n)$.

## 4 A PRACTICAL ALGORITHM

In practice, we do not know the exact correlation for each key (we may only know a much smaller set of high-frequency keys, just like the MCV (Most Common Values) optimization [39] in PostgreSQL). Besides, we also need to take into account the constrained memory budget. In this section, we introduce how we build a practical algorithm based on theorems/corollaries we obtained earlier when deriving the optimal algorithm.

### 4.1 Rounded Hash

We first propose **Rounded** Hash (RH) based on Theorem 3.4. One insight from Theorem 3.4 is that most partitions should be divisible by $c_R$ in the optimal partitioning. To understand why this insight may help, consider the example in Figure 5. When we have 5 pages in the buffer, we can produce at most four partitions since one page is used for streaming the input. In the top part of the figure, uniform hash partitioning creates 4 partitions with each having $18/4 = 4.5$ pages worth of data (for illustration, we fix the fudge factor as 1.0). As we discussed in the previous section, NBJ is mostly used in the probing phase due to read/write asymmetry. However, if we apply NBJ, the maximum #chunks we load at a time is 3-page long since we need one page for streaming the outer relation and one page for the output. As a result, in the example shown, for each of the four partitions, we will read the outer relation twice (i.e., 2 chunks per partition). However, we can avoid some of these accesses if we allow for non-uniform partitioning as shown at the bottom of the figure. We next discuss how to achieve this. Formally, uniform partitioning assigns records using

$$\text{PartID} = \text{hash(key)} \mod m \tag{5}$$

where $m$ is the number of partitions. Therefore, to reduce the extra I/O cost using non-uniform partitioning, we propose a **Rounded**

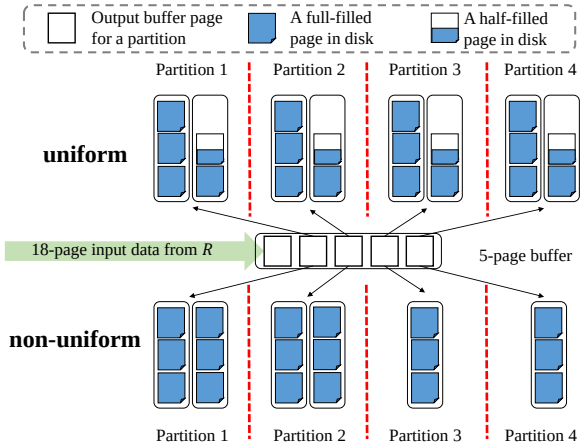



**Figure 5: With uniform partitioning, no partitions of $R$ fit in memory and thus all the records from $S$ are scanned twice. However, with non-uniform partitioning, only partitions 1 and 2 from $S$ are scanned twice.**

Hash (RH) as shown in Equation (6).

$$\text{PartID} = (\text{hash(key)} \mod \lceil n/c_R \rceil) \mod m \tag{6}$$

where $c_R = \lfloor b_R \cdot (B-2)/F \rfloor$ represents the chunk size in records, as defined earlier. In the example shown in Figure 5, the number of total chunks $n/c_R$ is 6 and the number of partitions $m$ is 4. Applying the *mod 6* operation, keys with a hash value remainder 0 or 4 are assigned to the first partition, keys with a hash value remainder 1 or 5 are assigned to the second partition and the remaining keys are evenly distributed in partitions 3 and 4. In that way, when joining partitions 1 and 2 with NBJ, we will perform two passes on the corresponding inner partitions, while partitions 3 and 4 will only lead to a single pass. In fact, we maximize the number of partitions that have $\lfloor \lceil n/c_R \rceil / (B-1) \rfloor$ chunks. In the above example, the number of partitions with $\lfloor (18/3)/4 \rfloor = 1$ chunk is maximized. In addition to reducing unnecessary passes, RH can also merge small partitions to avoid fragmentation when partitioning [25].

**Parametric Optimization.** Since we still use a hash function to distribute records, partitions 1 and 2 in Figure 5 could occasionally "overflow" and grow larger than six pages due to randomness. In a skewed workload, overflowed partitions could result in higher cost compared to the cost of uniform partitioning. In fact, a more robust scheme in Figure 5 is to assign 2.5 pages to partition 4 and evenly distribute 15.5 pages to partitions 1, 2, and 3. Formally, we replace $c_R$ with $c_R^* = \beta \cdot c_R$ in Equation (6), where $\beta$ should be very close to 1 in the range $(0, 1]$ (we fix $\beta = 0.95$ in our implementation).

**Cost Estimation.** To get a rough estimation of how many I/Os used by RH, we first need to know how to build the I/O model for plain hash (random partitioning). Here we assume NBJ is used for partition-wise joins (we also use textbook equations to estimate the cost of SMJ and GHJ in our implementation). When we use hash functions to distribute records, we do not have control over how many records there are in each partition due to the random process. For simplicity, we assume that assigning records into a random partition can be treated as a balls-into-bins problem, and thus the size of each partition can be approximated by a Poisson

distribution with $\lambda = \frac{n}{m}$, s.t., $\sum_{j=1}^{m} \|P_j\| = n$. As each partition has the same expected size with the plain hash function, the expected overall cost is given as follows:

$$\begin{aligned}\mathbb{E}[\mathsf{Join}(\mathbb{P}_{\mathsf{hash}})] &= \sum_{j=1}^{m} \sum_{i \in P_j} \mathsf{CT}[i] \cdot \mathbb{E}\left[\left\lceil \|P_j\|/c_R \right\rceil\right] \\ &\approx \sum_{j=1}^{m} \sum_{i \in P_j} \mathsf{CT}[i] \cdot \lceil n/(m \cdot c_R) \rceil \\ &= \lceil n/(m \cdot c_R) \rceil \cdot \sum_{i=1}^{n} \mathsf{CT}[i]\end{aligned} \tag{7}$$

We further generalize the above cost model of Plain Hash (PH) by defining the estimated probe cost of any consecutive subsequence between $s$ and $e$ in CT, noted by $g_{PH}(s, e, m)$:

$$g_{\mathsf{PH}}(s, e, m) = \lceil \frac{e - s + 1}{m \cdot c_R} \rceil \cdot \sum_{i=s}^{e} \mathsf{CT}[i] \tag{8}$$

When estimating the cost of Rounded Hash, we need to know how much proportion (noted by $\gamma$) of data from $S$ is scanned with one fewer pass ($1 - \gamma$ of data is scanner with one more pass):

$$\gamma = \left( (\lceil \frac{e - s + 1}{c_R^*} \rceil \bmod m) \cdot \lfloor \frac{e - s + 1}{m \cdot c_R^*} \rfloor \cdot c_R^* \right) \cdot 100\% / (e - s + 1)$$

As such, the normalized number of rounded passes is:

$$\#\mathsf{rounded\_passes} = \gamma \cdot \lfloor \frac{e - s + 1}{m \cdot c_R^*} \rfloor + (1 - \gamma) \cdot \lceil \frac{e - s + 1}{m \cdot c_R^*} \rceil$$

We then have our cost model for rounded hash $g_{\mathsf{RH}}$:

$$g_{\mathsf{RH}}(s, e, m) = \#\mathsf{rounded\_passes} \cdot \sum_{i=s}^{e} \mathsf{CT}[i] \tag{9}$$

**Overestimation using Chernoff Bound.** When the filling percent for each partition using plain hash reaches the predefined threshold $\beta$, ($\frac{e-s+1}{m} > \beta \cdot t \cdot c_R$, where $t$ is the largest positive integer such that $t \cdot c_R > \frac{e-s+1}{m}$), we disable rounded hash. This is to avoid additional passes that are triggered by occasional overflow in the random process. And we also need to take into account the possible overflow when estimating the cost. In fact, every element is distributed independently and identically using the same hash function, and thus we can apply Chernoff bound to overestimate the probability. For an arbitrary partition, we define $X = \sum_{i=s}^{e} X_i$ where $X_i = 1$ (i.e., ($i$-th element is assigned to this partition) with the probability $\frac{1}{m}$, and $X_i = 0$ with probability $1 - \frac{1}{m}$. Then $\mathbb{E}[X] = \frac{e-s+1}{m}$. From Chernoff bound, $\Pr[X > t \cdot c_R] < \left(\frac{e^\sigma}{(1+\sigma)^{1+\sigma}}\right)^{\mathbb{E}[X]}$, where $\sigma = \frac{t \cdot c_R \cdot m}{e-s+1} - 1$. We thus let $1 - \gamma = \left(\frac{e^\sigma}{(1+\sigma)^{1+\sigma}}\right)^{\mathbb{E}[X]}$ and set:

$$\#\mathsf{rounded\_passes} = \gamma \cdot \left\lceil \frac{e - s + 1}{m \cdot c_R^*} \right\rceil + (1 - \gamma) \cdot \left( \left\lceil \frac{e - s + 1}{m \cdot c_R^*} \right\rceil + 1 \right)$$

We then reuse Equation (9) to estimate the I/O cost.

**Algorithm 6:** HYBRID-PARTITION($HS_{\mathsf{mem}}, f_{\mathsf{disk}}, m_r$)

```
for every record_i = (key_i, value_i) from relation R do
    if key_i ∈ HS_mem then
        put record_i in the in-memory hash table HT_mem
    else
        if key_i ∈ f_disk then
            assign record_i to disk-resident partition f_disk(k)
        else
            use DHH/RH to partition record_i with m_r pages
```

## 4.2 Skew Optimization

PostgreSQL supports skew optimization by assigning 2% memory to build a hash table for skewed keys if their total frequency is larger than 1% of the outer relation. In this section, we introduce a more general version of skew optimization, inspired by Corollary 3.3. Our skew optimization does not require additional pre-processing because it can be built on the same input from PostgreSQL – a certain amount of Most Common Values (MCVs). Besides, we also discuss how we achieve a robust skew optimization without any predefined thresholds, and how we design our algorithm with enforcing memory constraints.

**Designated and Random Partitioning**. When we build the optimal partitioning $\mathbb{P}$ using Algorithm 5, we know exactly the partition assignment (i.e., the mapping function $f$) for each key. Although we cannot have the designated partition for each key in a practical algorithm, the output of the approximate partitioning should *partially* solve the partitioning problem. Specifically, we have a hash set ($HS_{\mathsf{mem}}$) to store the (skewed) keys that should be used to build the in-memory hash table, a hash map ($f_{\mathsf{disk}}$) that stores which keys should be assigned to specific partitions, and the rest of keys will be partitioned uniformly using either DHH or Rounded Hash (RH) with a certain memory budget in pages ($m_r$). Compared to PostgreSQL's skew optimization, $HS_{\mathsf{mem}}$ replaces the thresholds-based heuristic rules, and $f_{\mathsf{disk}}$ is a new element that guides the partitioning for partial data. In fact, when we have $HS_{\mathsf{mem}}$, $HS_{\mathsf{disk}}$ and $m_r$, we can partition $R$ using Algorithm 6. The new partitioning workflow is as follows: for each record in $R$, if its key is in $HS_{\mathsf{mem}}$, we put it in the in-memory hash table $HT_{\mathsf{mem}}$; if its key belongs to $f_{\mathsf{disk}}$, we put it into the partition specified by $f_{\mathsf{disk}}$; otherwise, it participates in a DHH/RH (along with the rest of the records) with $m_r$ pages. The criterion to trigger DHH is $\lceil n'/c_R^* \rceil < m_r$. This is because, when $\lceil n'/c_R^* \rceil < m_r$ (where $n' = n_R - |HS_{\mathsf{mem}}| - |f_{\mathsf{disk}}|$ is the number of records that do not belong to $HS_{\mathsf{mem}}$ or $f_{\mathsf{disk}}$), RH makes no difference according to Equation (6), and we could exploit the available memory with HHJ when $m_r$ is large.

**Enforcing Memory Constraints.** State-of-the-art systems (e.g., MySQL and PostgreSQL) typically assign to each join operator a user-defined memory budget. For example, the default memory limit for the join operator in MySQL is 256 KB [33] and the one in PostgreSQL is 4 MB [40]. Assuming the available buffer is in total $B$ pages, we need to ensure that Algorithm 6 uses no more than $B$ pages. We present a memory breakdown in Figure 6. The number of pages for the in-memory hash table $B_{HT}(k_{\mathsf{mem}})$ is linear

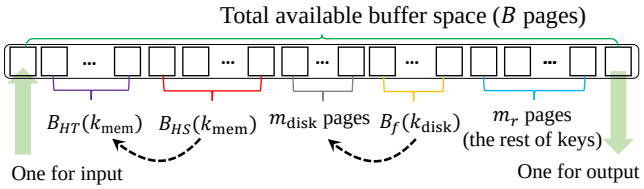



**Figure 6: The memory breakdown when partitioning $R$. $B_{HT}(k_{\text{mem}}), B_{HS}(k_{\text{mem}}), B_f(k_{\text{disk}})$ represent the number of pages occupied by a hash table for records, a hash set for skew keys, and a hash map between keys and their partition ids. $m_{\text{disk}}$ pages are reserved for the partitions in $f_{\text{disk}}$ and $m_r$ pages are used to partition the remaining keys.**

to $k_{\text{mem}}$, the number of keys in $HS_{\text{mem}}$. We also note $B_{HS}(k_{\text{mem}})$ as the number of pages of $HS_{\text{mem}}$. Similarly, we note by $B_f(k_{\text{disk}})$ the number of pages of the hash map $f$ with $k_{\text{disk}}$ keys. With key size as $ks$, the page size as $ps$, we may roughly have, $B_{HT}(k_{\text{mem}}) = \lceil b_R \cdot k_{\text{mem}}/F \rceil$, $B_{HS}(k_{\text{mem}}) = \lceil ks \cdot k_{\text{mem}}/(F \cdot ps) \rceil$, and $B_f(k_{\text{disk}}) = \lceil (ks+4) \cdot k_{\text{disk}}/(F \cdot ps) \rceil$, where a 4-byte integer is used to store the partition id in the hash map $f$. The memory constraint can then be rewritten as:

$$B_{HT}(k_{\text{mem}}) + B_{HS}(k_{\text{mem}}) + B_f(k_{\text{disk}}) + m_{\text{disk}} + m_r \le B - 2 \quad (10)$$

where $k_{\text{mem}} + k_{\text{disk}} \le k$, and $k$ is the number of tracked high-frequency keys. Using $HS_{\text{mem}}$ and $f_{\text{disk}}$, we can designate the partitions for the top $k_{\text{mem}} + k_{\text{disk}}$ keys from the tracked key list.

**Algorithm 7:** NOCAP(CT, $k, n, m$)

```
1  c_opt ← +∞, k_mem, f_disk, k_disk;
2  for i_1 ← 0 to min(k, c_R) do
3      V ← Partition(CT[i_1 + 1 : k], k − i_1, ⌈(k − i_1)/c_R⌉);
4      for i_2 ← 0 to min(k, c_R) − i_1 do
5          for j ← min(i_2, 1) to ⌈i_2/c_R⌉ do
6              m_r ← B − 2 − B_HT(i_1) − B_HS(i_1) − B_f(i_2) − j;
7              c_probe ← V[i_2][j].cost;
8              c_part ← μ · (⌈i_2/b_R⌉ + ⌈(PCT[i_2+i_1] − PCT[i_1])/b_S⌉);
9              c_rest ← g_RH/DHH(i_2 + 1, n, m_r);
10             c_tmp ← c_probe + c_part + c_rest;
11             if c_tmp < c_opt then
12                 c_opt ← c_tmp, k_mem ← i_1, k_disk ← i_2;
13                 f_disk ← GetCut(V, i_2, j);
14 return k_mem, k_disk, f_disk;
```

**Tuning $k_{\text{mem}}$, $k_{\text{disk}}$ and $f_{\text{disk}}$.** To find out the best combination of $k_{\text{mem}}$, $k_{\text{disk}}$ and $f_{\text{disk}}$ ($m_r$ can be calculated using Equation (10)), we iterate all possible combinations of them, and pick the one with the minimum estimated cost. We summarize this procedure in Algorithm 7. When estimating the overall join cost, we can have an accurate estimation for $k_{\text{disk}}$ keys, which we know the exact frequency from the top-$k$ list, but we can only estimate the join cost for the rest of keys $n - k_{\text{mem}} - k_{\text{disk}}$ using $m_r$ pages. In line 9 of Algorithm 7, we use $g_{\text{RH/DHH}}(k_{\text{mem}} + k_{\text{disk}} + 1, n, m_r)$ to represent the estimated cost for the rest of keys. The partitioning method can be either RH or DHH, depending on how large $m_r$ is, as mentioned earlier. For DHH, we configure $m_{\text{DHH}}$ as mentioned in Section 2.2. We can roughly estimate how many records are cached when $m_{\text{DHH}}$ is given, and then we can reuse the model for HHJ in Section 3.2 to estimate #I/Os. Note that, in both estimation models of DHH and RH, we need to calculate $\sum_{i=k_{\text{mem}}+k_{\text{disk}}+1}^{n} \text{CT}[i]$. Although we do not have any info about $\text{CT}[i]$ for these keys, we can estimate this summation by $n_S - \sum_{i=1}^{k_{\text{mem}}+k_{\text{disk}}} \text{CT}[i] = n_S - \text{PCT}[k_{\text{mem}} + k_{\text{disk}}]$ where $n_S$ is the estimated number of records in relation $S$.

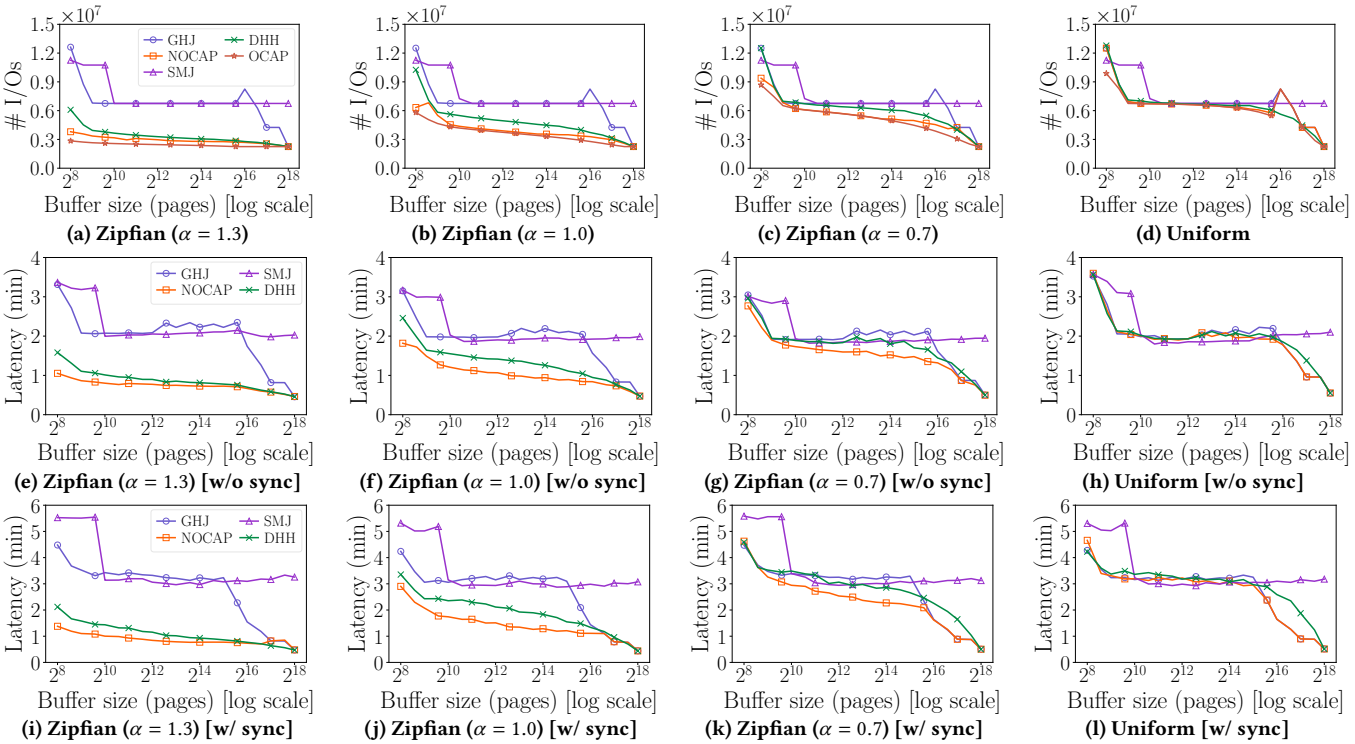
## 5 EXPERIMENTAL ANALYSIS

We now present our experimental results for the methods introduced in Sections 3 and 4.

**Experimental Setup.** We use our in-house server, which is equipped with two Intel Xeon Gold 6230 2.1GHz processors, each having 20 cores with virtualization enabled. For our storage, we use a PCIe P4510 SSD with direct I/O enabled. We can vary the read/write asymmetry by turning the `O_SYNC` flag on and off. When `O_SYNC` is on, every write is ensured to flush to disk before the write is completed, and thus has higher asymmetry ($\mu_{sync} = 5$, $\tau_{sync} = 3.5$). When it is off, we have lower asymmetry ($\mu_{no\_sync} = 2.9$, $\tau_{no\_sync} = 2.1$). Unless otherwise specified, sync I/O is off to accelerate joins. All the approaches are implemented in C++, compiled with gcc 10.1.0, in a CentOS Linux with kernel 4.18.0.

**Approaches Compared.** We compare NOCAP with Grace Hash Join (GHJ), Sort-Merge Join (SMJ), and Dynamic Hybrid Hash join (DHH). For all the partitioned-based methods (GHJ, DHH, and ours), we apply a light optimizer that picks the most efficient algorithm according to Table 1 in the partition-wise join. We also augment GHJ by allowing it to fall back to NBJ if the latter has a lower cost. For DHH, we follow prior approaches that use fixed thresholds to trigger skew optimization (3% of the available memory is used for an in-memory hash table for skew keys if the sum of their frequency is larger than 1% of the relation size). For NOCAP and DHH, we assume top $k = 50K$ frequently matching keys are tracked when $n = 1M$ (i.e., 5% of the keys). In PostgreSQL's implementation of DHH, the frequent keys are stored using a small amount of system cache, which does not compete with the user-defined working memory unless these keys are used to build the in-memory hash table. Similarly, in our emulation, we also assume that the top-$k$ keys are given. Due to the pruning techniques (§3.1.4), computing the approximate partition scheme (Algorithm 7) with $k = 50K$ takes less than one second, so we omit the discussion of its runtime.

**Experimental Methodology.** We experiment with a synthetic benchmark and with TPC-H. For the synthetic benchmark, we emulate a PK-FK equi-join between two tables $R$ and $S$ with $n_R = 1M$ and $n_S = 8M$. The record size is 1KB for both $R$ and $S$, and thus we have $\|R\| = 250K$ pages and $\|S\| = 2M$ pages. In our experiments, we vary the buffer size from $0.5 \cdot \sqrt{F \cdot \|R\|} \approx 256$ pages to $\|R\| = 250K$ pages. We use #I/Os (reads + writes) and latency as two metrics, and when comparing #I/Os, we also run OCAP (§3.2) to plot the optimal (lower bound) #I/Os .




(a) Zipfian ($\alpha = 1.3$) (b) Zipfian ($\alpha = 1.0$) (c) Zipfian ($\alpha = 0.7$) (d) Uniform

(e) Zipfian ($\alpha = 1.3$) [w/o sync] (f) Zipfian ($\alpha = 1.0$) [w/o sync] (g) Zipfian ($\alpha = 0.7$) [w/o sync] (h) Uniform [w/o sync]

(i) Zipfian ($\alpha = 1.3$) [w/ sync] (j) Zipfian ($\alpha = 1.0$) [w/ sync] (k) Zipfian ($\alpha = 0.7$) [w/ sync] (l) Uniform [w/ sync]

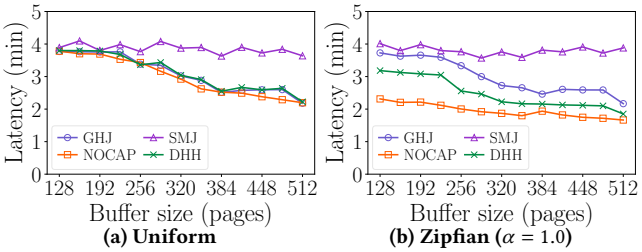
**Figure 7: While state-of-the-art skew optimization in DHH helps reduce the I/O cost, NOCAP can better exploit the correlation skew to achieve even lower I/O cost and latency. The benefit of NOCAP is more pronounced with skew correlation.**

## 5.1 Experimental Results

**NOCAP Dominates for any Correlation Skew and any Memory Budget.** Figures 7a, 7b, 7c, and 7d show that NOCAP achieves the lowest I/O cost (near-optimal) among all the join algorithms for any correlation skew and any memory budget, except for a bump after $2^{16}$ pages in Figure 7d. We also observe a similar I/O spike for GHJ in all the above figures. This is because GHJ and NOCAP detects that NBJ with higher #I/Os can lead to lower latency compared to partitioning-based joins because of read/write asymmetry (i.e., $\mu > 1$, writes are generally slower than reads). To verify this, we see the latency of GHJ and NOCAP still decreases after $2^{16}$ pages in Figures 7e, 7f, 7g, and 7h, even though #I/Os increases. In addition to the spike for GHJ and NOCAP when the buffer has more than $2^{16}$ pages, all the join algorithms generally have fewer #I/Os and thus lower latency as the buffer size increases. We observe a latency spike for GHJ between $2^{12}$ and $2^{16}$ pages, due to garbage collection in the SSD. When implementing GHJ, we use $m = \lceil n/c_R^* \rceil$ to avoid fragmented partitions, and thus GHJ will have fewer partitions when $B$ increases ($c_R^* = \beta \cdot \lfloor b_R \cdot (B-2)/F \rfloor$). Fewer partitions imply more frequent writes when partitioning. If sync I/O is on, there could be more garbage collection in SSDs and thus higher latency per write (around $10\mu s$ more per write). We verify this by re-running the experiments with disabling sync I/O. Observe from Figures 7i, 7j, 7k and 7l, we have a much smaller spike for GHJ when we increase the buffer size.

When we compare Figures 7a, 7b, 7c, and 7d, we also conclude that GHJ and SMJ have nearly the same I/O pattern when we vary the correlation skew, while DHH and NOCAP can take advantage of correlation skew to alleviate #I/Os. However, DHH cannot fully exploit the data skew because it limits the space for high-frequency keys with a fixed threshold (3% of the total memory budget). Compared to DHH, NOCAP normally has fewer #I/Os because it freely decides the size of the in-memory hash table for frequent keys without any predefined thresholds. Figures 7f-7h show that the latency gap between NOCAP and DHH is more pronounced when we have higher skew ($\alpha = 1$ or $\alpha = 0.7$), compared to the uniform correlation. However, when the correlation is highly skewed ($\alpha = 1.3$), the small hash table (3% of the available memory) employed by DHH is enough to capture the most frequent keys and, hence, issue near-optimal #I/Os (Figure 7a). As a result, the benefit of NOCAP for $\alpha = 1.3$ (Figure 7e) is smaller than for $\alpha = 1$ (Figure 7f). Nevertheless, NOCAP still issues 30% fewer #I/Os when the memory budget is small ($2^8$ pages). In addition, for uniform correlation, NOCAP has higher latency compared to SMJ when the buffer size $B \in [2^{12}, 2^{14}]$. Although #I/Os is similar (Figure 7d), SMJ has slightly lower latency than all other partitioning-based join algorithms (Figure 7h) because the majority of writes during partitioning are random, while during sort-merging sequential, thus faster (i.e., $\tau < \mu$).

**NOCAP Outperforms DHH Even for Uniform Correlation Under Low Memory Budget.** We also examine the speedup when we have a lower memory budget. Although Figures 7l and 7h show that there *seems* no difference between DHH and NOCAP when the workload is uniform, we actually capture a larger difference after we narrow down the buffer range (128~512 pages). As shown in Figure 8a, NOCAP can even achieve up to 15% and 10% speedup when there are 480 and 352 pages. The step-wise pattern of DHH



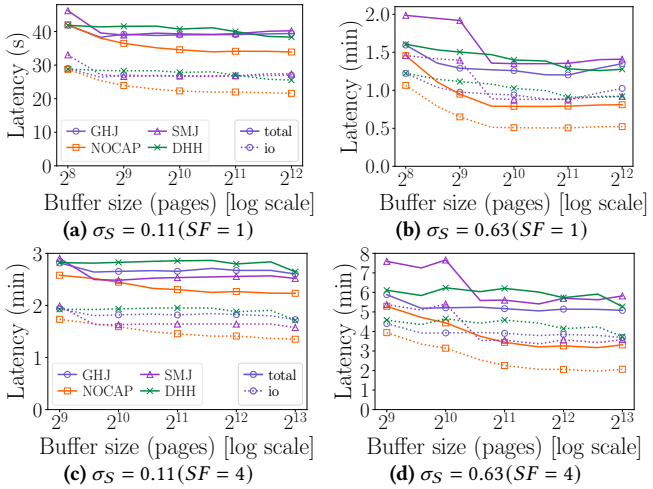

**(a) Uniform** **(b) Zipfian ($\alpha = 1.0$)**

**Figure 8: When the memory is limited, NOCAP can reach 10% speedup even if the workload is uniform.**

and GHJ comes from random (uniform) partitioning. When the memory is smaller than $\sqrt{\|R\| \cdot F}$, uniform partitioning easily makes each partition larger than the chunk size, as illustrated in Figure 5. On the contrary, rounded hash allows some partitions to be larger so that other partitions have fewer chunks and thus require fewer passes to complete the partition-wise joins. When the correlation becomes more skewed, NOCAP can be more than 30% faster than DHH under limited memory, as shown in Figure 8b.

**Experiments with TPC-H Data.** We experiment with a modified TPC-H Q12. Q12 selects data from table `lineitem`, and joins them with `orders` on `l_orderskey=o_orderskey`, followed by an aggregation. As we have already seen, DHH has only a minor difference from NOCAP when the correlation is uniform. Here, we focus on a skew correlation in TPC-H data. To achieve this, we modify the TPC-H dbgen codebase – all the keys are classified into *hot* or *cold* keys, and the frequency of hot keys and cold keys, respectively, follows two different uniform distributions, which controls the overall skew and the average matching keys [11, 13]. In this experiment, we focus on a skew correlation where 0.75% of the `o_orderkey` keys match 1600 records in `lineitem` and the rest keys only match 4 records on average. The above configuration results in 4× larger `lineitem` table than the origin `lineitem`. When the scale factor (SF) is 1, $\sqrt{\|\texttt{orders}\| \cdot F} \approx 264$ pages. We vary the buffer size from $256 = 2^8$ pages to $2^{12}$ pages when the scale factor $SF = 1$. We further remove the filtering condition on `l_shipmode` and `l_receiptdate` so that the size of the filtered `lineitem` is comparable to the size of `orders` (we focus on the case when $\|R\| < \|S\|$). After removing these two conditions, we have selectivity $\sigma_S \approx 0.11$ where $S$ refers to `lineitem`. If we further remove another filtering condition `l_shipdate < l_commitdate`, we have $\sigma_S \approx 0.63$. We run our experiments with these two levels of selectivity and two scale factors ($SF = 1$, $SF = 4$), as shown in Figure 9.

**NOCAP Accelerates TPC-H Joins.** We observe that the difference in the total latency between NOCAP and DHH is not so large as we see in previous experiments on skew correlation in Figure 9a. In fact, we find that the proportion of time spent on I/Os is much lower in the TPC-H experiment due to extra aggregations. In earlier experiments (e.g., Figures 7f and 7g), the time spent on CPU is 10~20 seconds out of a few minutes (the total latency). In contrast, we observe that in both Figures 9a and 9b, the proportion of time spent on I/Os is at most 75%. This explains why the speedup of NOCAP diminishes when we have more CPU-intensive operations (e.g., aggregation) in a query. In addition, when we adjust the selectivity so that more data from `lineitem` are involved in the join, we observe a higher benefit in Figure 9b. We have similar observation




**(a)** $\sigma_S = 0.11 (SF = 1)$ **(b)** $\sigma_S = 0.63 (SF = 1)$

**(c)** $\sigma_S = 0.11 (SF = 4)$ **(d)** $\sigma_S = 0.63 (SF = 4)$

**Figure 9: TPC-H: NOCAP leads to higher speedup when more data from $S$ are joined in a query.**

when we move to a larger data set (SF=4), as shown in Figures 9c and 9d. In fact, in both cases, when we have larger `lineitem` data involved in the join, NOCAP usually leads to higher speedup.

## 6 RELATED WORK

**In-Memory Joins.** When the memory is enough to store the entire relations, in-memory join algorithms can be applied [5, 6, 30, 38, 44, 46]. Similar to storage-based joins, in-memory join algorithms can be classified as hash-based and sort-based (e.g., MPSM [1] and MWAY [3]). Hash joins can be further classified into partitioned joins (e.g., parallel radix join [3]) and non-partitioned joins (e.g., CHT [4]). GPUs can also be applied to accelerate in-memory joins [18, 31, 43]. Although storage-based joins focus on #I/Os, when two partitions both fit in memory after the partitioning phase, NOCAP can still benefit from optimized in-memory joins.

**Distributed Equi-Joins.** In a distributed environment, the efficiency of equi-joins also relies on load balance and communication cost. When the join correlation distribution is skewed, uniform partitioning may overwhelm some workers by assigning excessive work to do [48]. More specifically, when systems like Spark [51] rely on in-memory computations, the overall performance drops when the transmitted data for a machine do not fit in memory. Several approaches [7, 28, 50] are proposed to tackle the data (and thus, workload) skew for distributed equi-joins.

## 7 CONCLUSIONS

In this paper, we propose a new cost model for partitioning-based PK-FK joins that allows us to find the optimal partitioning strategy assuming accurate knowledge of the join correlation. Using this optimal partitioning strategy, we show that the state-of-the-art Dynamic Hybrid Hash Join (DHH) yields sub-optimal performance since it does not fully exploit the available memory and the correlation skew. To address DHH's limitations, we develop a practical near-optimal partitioning-based join with variable memory requirements (NOCAP) which is based on OCAP. We show that NOCAP dominates the state of the for any available memory budget and for any join correlation, leading to up to 30% improvement.

# REFERENCES


[1] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *Proceedings of the VLDB Endowment* 5, 10 (2012), 1064–1075. https://doi.org/10.14778/2336664.2336678

[2] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1905–1916. https://doi.org/10.14778/2733085.2733096

[3] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96. https://doi.org/10.14778/2732219.2732227

[4] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Ozsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[5] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 168–180. https://doi.org/10.1145/3448016.3452831

[6] Ronald Barber, Guy M Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi K Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-Efficient Hash Joins. *Proceedings of the VLDB Endowment* 8, 4 (2014), 353–364. https://doi.org/10.14778/2735496.2735499

[7] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in parallel query processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*. 212–223. https://doi.org/10.1145/2594538.2594558

[8] Anna Berenberg and Brad Calder. 2021. Deployment Archetypes for Cloud Applications. *CoRR* abs/2105.0 (2021). https://arxiv.org/abs/2105.00560

[9] Spyros Blanas, Yinan Li, and Jignesh M Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 37–48. https://doi.org/10.1145/1989323.1989328

[10] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 54–65. http://www.vldb.org/conf/1999/P5.pdf

[11] Jehoshua Bruck, Jie Gao, and Anxiao Jiang. 2006. Weighted Bloom filter. In *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*. 2304–2308. https://doi.org/10.1109/ISIT.2006.261978

[12] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo N Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco A S Netto, Adel Nadjaran Toosi, Maria Alejandra Rodriguez, Ignacio Mart\'\in Llorente, Sabrina De Capitani di Vimercati, Pierangela Samarati, Dejan S Milojicic, Carlos A Varela, Rami Bahsoon, Marcos Dias de Assunção, Omer F Rana, Wanlei Zhou, Hai Jin, Wolfgang Gentzsch, Albert Y Zomaya, and Haiying Shen. 2019. A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade. *Comput. Surveys* 51, 5 (2019), 105:1—-105:38. https://doi.org/10.1145/3241737

[13] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2022. Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. In *In Proceedings of the Very Large Databases Endowment*. https://doi.org/10.14778/3485450.3485461

[14] Cisco. 2018. Cisco Global Cloud Index: Forecast and Methodology, 2016–2021. *White Paper* (2018). https://virtualization.network/Resources/Whitepapers/0b75cf2e-0c53-4891-918e-b542a5d364c5{_}white-paper-c11-738085.pdf

[15] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1–8. http://dl.acm.org/citation.cfm?id=602259.602261

[16] David J DeWitt, Jeffrey F Naughton, Donovan A Schneider, S Seshadri, Wei Li, Dengfeng Gao, Richard T Snodgrass, Kien A Hua, and Chiang Lee. 1992. Practical Skew Handling in Parallel Joins. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 525–535. https://doi.org/10.1145/564691.564711

[17] Gartner. 2017. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. https://tinyurl.com/Gartner2020.

[18] Chengxin Guo and Hong Chen. 2019. In-Memory Join Algorithms on GPUs for Large-Data. In *21st IEEE International Conference on High Performance Computing and Communications; 17th IEEE International Conference on Smart City; 5th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2019, Zhangjiajie, China, August 10-12*. 1060–1067. https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00151

[19] Laura M Haas, Michael J Carey, Miron Livny, and Amit Shukla. 1997. Seeking the Truth About ad hoc Join Costs. *The VLDB Journal* 6, 3 (1997), 241–256. https://doi.org/10.1007/s007780050043

[20] Shuo He, Xinchen Lyu, Wei Ni, Hui Tian, Ren Ping Liu, and Ekram Hossain. 2020. Virtual Service Placement for Edge Computing Under Finite Memory and Bandwidth. *IEEE Trans. Commun.* 68, 12 (2020), 7702–7718. https://doi.org/10.1109/TCOMM.2020.3022692

[21] Kien A Hua and Chiang Lee. 1991. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 525–535. http://www.vldb.org/conf/1991/P525.PDF

[22] Shiva Jahangiri, Michael J Carey, and Johann-Christoph Freytag. 2022. Design Trade-offs for a Robust Dynamic Hybrid Hash Join. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2257–2269. https://www.vldb.org/pvldb/vol15/p2257-jahangiri.pdf

[23] Brendan Jennings and Rolf Stadler. 2015. Resource Management in Clouds: Survey and Research Challenges. *J. Netw. Syst. Manag.* 23, 3 (2015), 567–619. https://doi.org/10.1007/s10922-014-9307-7

[24] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak Borkar, Yingyi Bu, Michael J. Carey, Murtadha Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, Chen Luo, Ian Maxon, and Pouria Pirzadeh. 2020. Robust and efficient memory management in Apache AsterixDB. *Software - Practice and Experience* 50, 7 (2020), 1114–1151. https://doi.org/10.1002/spe.2799

[25] Masaru Kitsuregawa, Masaya Nakayama, and Mikio Takagi. 1989. The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 257–266. http://www.vldb.org/conf/1989/P257.PDF

[26] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. 1983. Application of Hash to Data Base Machine and Its Architecture. *New Generation Computing* 1, 1 (1983), 63–74.

[27] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[28] Rundong Li, Mirek Riedewald, and Xinyan Deng. 2018. Submodularity of Distributed Join Computation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1237–1252. https://doi.org/10.1145/3183713.3183728

[29] Wei Li, Dengfeng Gao, and Richard T Snodgrass. 2002. Skew handling techniques in sort-merge join. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 169–180. https://doi.org/10.1145/564691.564711

[30] Feilong Liu and Spyros Blanas. 2015. Forecasting the cost of processing multi-join queries via hashing for main-memory databases (Extended version). *CoRR* abs/1507.0 (2015). http://arxiv.org/abs/1507.03049

[31] Clemens Lutz, Sebastian Breß and Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1633–1649. https://doi.org/10.1145/3318464.3389705

[32] John C. McCallum. 2022. Historical Cost of Computer Memory and Storage. *https://jcmit.net/mem2015.htm* (2022).

[33] MySQL. 2021. MySQL System Variables. *https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html* (2021).

[34] MySQL. 2023. MySQL. *https://www.mysql.com/* (2023).

[35] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. 1988. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 468–478. http://www.vldb.org/conf/1988/P468.PDF

[36] Tarikul Islam Papon and Manos Athanassoulis. 2021. A Parametric I/O Model for Modern Storage Devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*.

[37] Tarikul Islam Papon and Manos Athanassoulis. 2021. The Need for a New I/O Model. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[38] Constantin Pohl, Kai-Uwe Sattler, and Goetz Graefe. 2020. Joins on high-bandwidth memory: a new level in the memory hierarchy. *The VLDB Journal* 29, 2-3 (2020), 797–817. https://doi.org/10.1007/s00778-019-00546-z

[39] PostgreSQL. [n. d.]. Skew Optimization in PostgreSQL. *https://github.com/postgres/postgres/blob/master/src/include/executor/hashjoin.h#L84* ([n. d.]).

[40] PostgreSQL. 2022. PostgreSQL Resource Consumption. *https://www.postgresql.org/docs/13/runtime-config-resource.html* (2022).

[41] PostgreSQL. 2023. PostgreSQL: The World's Most Advanced Open Source Relational Database. *https://www.postgresql.org* (2023).

[42] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition.

[43] Ran Rui, Hao Li, and Yi-Cheng Tu. 2015. Join algorithms on GPUs: A revisit after seven years. In *Proceedings of the IEEE International Conference on Big Data (BigData)*. 2541–2550. https://doi.org/10.1109/BigData.2015.7364051

[44] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1961–1976. https://doi.org/10.1145/2882903.2882917

[45] Leonard D Shapiro. 1986. Join Processing in Database Systems with Large Main Memories. *ACM Trans. Database Syst.* 11, 3 (1986), 239–264. https://doi.org/10.1145/6314.6315

[46] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 510–521. http://dl.acm.org/citation.cfm?id=645920.758363

[47] TPC. 2021. TPC-H benchmark. *http://www.tpc.org/tpch/* (2021).

[48] Christopher B Walton, Alfred G Dale, and Roy M Jenevein. 1991. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 537–548. http://www.vldb.org/conf/1991/P537.PDF

[49] Jan Wassenberg and Peter Sanders. 2011. Engineering a Multi-core Radix Sort. In *Proceedings of the International Conference on Parallel Processing (EuroPar)*. 160–169. https://doi.org/10.1007/978-3-642-23397-5_16

[50] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. 2008. Handling data skew in parallel joins in shared-nothing systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1043–1052. https://doi.org/10.1145/1376616.1376720

[51] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

## A WHAT IF $S$ IS SMALLER

We still assume *NBJ* is mostly used for partition-wise joins, and we now consider the case if $S$ is smaller. Although $S$ serves as a fact table, and thus is usually larger than the dimension table $R$, it usually happens that we have some projection operators in $S$, which may only ask for a few attributes from $S$, and the record size of $S$ becomes quite smaller after we push down the projection operator for $S$ before executing the join operator. Now we load table $S$ into the memory chunk by chunk and scan $R$ for multiple times. The $\mathsf{NBJ}(R_j, S_j)$ can then be formulated by swapping $\|R_j\|$ and $\|S_j\|$:

$$\mathsf{NBJ}(R_j, S_j) = \|S_j\| + \left\lceil \frac{\|S_j\|}{(B-2)/F} \right\rceil \cdot \|R_j\|$$

By summing up the above cost for every pair of partitions, we have:

$$\begin{aligned}\mathsf{Join}(\mathbb{P}) &= \sum_{j=1}^{m} \left( \|S_j\| + \left\lceil \frac{\|S_j\|}{(B-2)/F} \right\rceil \cdot \|R_j\| \right) \\ &= \sum_{j=1}^{m} \left\lceil \sum_{i=1}^{n} \mathbb{P}_{i,j} \cdot \mathsf{CT}[i]/c_S \right\rceil \cdot \left( \sum_{i=1}^{n} \mathbb{P}_{i,j} \right) + \|S\| \\ &= \sum_{j=1}^{m} \left\lceil \sum_{i \in P_j} \mathsf{CT}[i]/c_S \right\rceil \cdot \|P_j\| + \|S\|\end{aligned}$$

where $c_S$ represents the number of records per chunk for relation $S$. Note that we cannot directly re-apply Lemma 3.1 in above cost function, because if we want to swap the partition assignment of two elements $i_1$ and $i_2$, we now need to swap $\mathsf{CT}[i_1]$ with $\mathsf{CT}[i_2]$, and we cannot predict how $\lceil \sum \mathsf{CT}[i]/c_S$ changes. Thus, we turn to find an upper bound of $\mathsf{Join}(\mathbb{P})$ to minimize, so as to indirectly reduce $\mathsf{Join}(\mathbb{P})$:

$$\mathsf{Join}(\mathbb{P}) < \sum_{j=1}^{m} \left( \sum_{i \in P_j} \mathsf{CT}[i]/c_S + 1 \right) \cdot \|P_j\| + \|S\| = \mathsf{Join}_{UB}(\mathbb{P})$$

Since we can treat $c_S$, $\sum_{j=1}^{m} \|P_j\|$, $\|S\|$ as fixed value, no matter how the partitioning scheme changes, we are now minimizing $\sum_{j=1}^{m} \sum_{i \in P_j} \mathsf{CT}[i] \cdot \|P_j\|$, where we can apply Lemma 3.1 and thus consecutive theorem (Theorem 3.2) also holds.

We can still use the dynamic programming framework in Algorithm 2 by changing CalCost function:

$$\mathsf{CalCost}(s, e) = (e - s + 1) \cdot \lceil \frac{\mathsf{PCT}[e] - \mathsf{PCT}[s-1]}{c_S} \rceil$$

## B COMPLEXITY ANALYSIS FOR MONOTONICITY-BASED PRUINING

Based on two pruning inequalities (3) and (4) from corollary 3.3, the range for $k$ is at most $\frac{i}{j} - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1$. If $\frac{i}{j} - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1 > \frac{i}{j}$, we can even skip the specific $j$ (line 6 in Algorithm 2) because we do not have to iterate any $k$ in this case. Therefore, the complexity for the third loop (line 8 $\sim$ 12) is

$$\max\left(\frac{i}{j} - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1, 0\right)$$

. Summing up the complexity for all possible $i$ and $j$, we have:

$$\begin{aligned}&\sum_{i=1}^{n} \sum_{j=2}^{m-1} \max\left(\frac{i}{j} - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1, 0\right) \\ &\leq \sum_{i=1}^{n} \sum_{j=2}^{m-1} \max\left(\frac{i}{j} - \frac{n-i}{m-j}, 0\right) = \sum_{j=2}^{m-1} \sum_{i=\frac{j \cdot n}{m}}^{n} \frac{i \cdot m - n \cdot j}{j \cdot (m-j)} \\ &= \sum_{j=2}^{m-1} \frac{\frac{n \cdot j + n \cdot m}{2} \cdot \left(n - \frac{n \cdot j}{m} + 1\right) - n \cdot j \left(n - \frac{n \cdot j}{m} + 1\right)}{j \cdot (m-j)} \\ &= \sum_{j=2}^{m-1} \frac{\frac{n \cdot (m-j)}{2} \cdot \left(n - \frac{n \cdot j}{m} + 1\right)}{j \cdot (m-j)} = \sum_{j=2}^{m-1} \frac{n \cdot \left(n - \frac{n \cdot j}{m} + 1\right)}{2j} \\ &= \frac{1}{2} \left( \sum_{j=2}^{m-1} n^2 \cdot \left(\frac{1}{j} - \frac{1}{m}\right) + \sum_{j=2}^{m-1} \frac{n}{j} \right) = O(n^2 \log m)\end{aligned} \tag{11}$$

As shown in Equation (11), monotonicity-based pruning reduces the time complexity to $O(n^2 \log m)$.

## C PROOF FOR STEP-BASED PRUNING

The proof for the divisible theorem (Theorem 3.4) is not trivial because it may conflict with the sort theorem (Theorem 3.3), i.e., these two theorems may give us two different partitioning strategies that have the same – optimal – cost. In other words, an optimal solution might satisfy either one of the two or both. For example, consider $n = c_R \cdot (B-2) + k_r$ $(0 < k_r < c_R)$. According to the sort theorem (Theorem 3.3), we will group the last $k_r$ records to form the smallest partition and the rest of records form $B-2$ partitions. However, Theorems 3.2 and 3.4 show that we need to group the first $k_r$ records together. Even though we may end up having a different partitioning scheme, the optimal cost remains the same. We now present a more relaxed version of 3.3, which we will use to prove Theorem 3.4.

Theorem C.1 (Weak Sort Theorem). *There exists an optimal partitioning $\mathbb{P} = (P_1, P_2, \cdots, P_m)$, where each $P_i$ contains consecutive records from sorted CT array, and $\lceil \|P_1\|/c_R \rceil \geq \lceil \|P_2\|/c_R \rceil \geq \cdots \geq \lceil \|P_m\|/c_R \rceil$.*

We omit this proof since it is similar to Corollary 3.3. Based on the optimal solution $P_{opt}$ from Corollary C.1, we can prove Theorem 3.4 as follows.

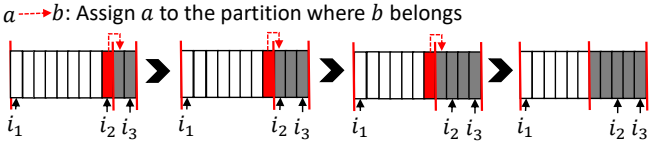
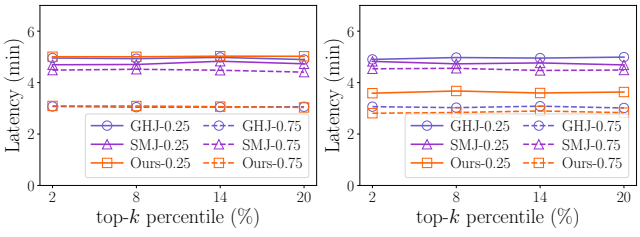



**Figure 10: When $\|\mathcal{P}_j\|$ is not divisible by $c_R$, we can move elements from $\mathcal{P}_{j-1}$ to $\mathcal{P}_j$ until $\left\lceil \frac{\|\mathcal{P}_{j-1}\|}{c_R} \right\rceil < \left\lceil \frac{\|\mathcal{P}_j\|}{c_R} \right\rceil$ or $\|\mathcal{P}_j\|$ is divisible by $c_R$. In the above example, we note the partition containing $[i_2+1, i_3]$ as $\mathcal{P}_j$ and the previous partition containing $[i_1, i_2]$ as $\mathcal{P}_{j-1}$. Assuming $c_R = 5$, the above process stops when $\|\mathcal{P}_j\| = k \cdot 5$) where $k \in \mathbb{N}$.**

Proof. We aim to find a process that can obtain another optimal solution which satisfies both Theorem 3.4 and Corollary C.1. We note $\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_m$ as the partition sequence obtained from Corollary C.1. As such, these partitions $\{\mathcal{P}_j | j \in m\}$ are sorted by $\left\lceil \frac{\|\mathcal{P}_j\|}{c_R} \right\rceil$ in a descending order. Now we select the largest $j$ where $\|\mathcal{P}_j\|$ is not divisible by $c_R$. If $j = 1$, we do nothing as the partitioning scheme already satisfies Theorem 3.4. Otherwise, we move adjacent elements from $\mathcal{P}_{j-1}$ to $\mathcal{P}_j$ until $\left\lceil \frac{\|\mathcal{P}_{j-1}\|}{c_R} \right\rceil < \left\lceil \frac{\|\mathcal{P}_j\|}{c_R} \right\rceil$ or $\|\mathcal{P}_j\|$ is a multiple of $c_R$. In this process, when we move elements to $\mathcal{P}_j$, $\left\lceil \frac{\|\mathcal{P}_j\|}{c_R} \right\rceil$ does not change since $\|\mathcal{P}_j\|$ is not divisible by $c_R$. Therefore, after we move one element $i$ to $\mathcal{P}_j$, all the elements from $\mathcal{P}_j$ contribute the same cost and all the elements (except $i$) from $\mathcal{P}_{j-1}$ have equivalent or even lower cost since $\|\mathcal{P}_{j-1}\|$ decreases. For element $i$, the cost cannot increase since we maintain the descending order of $\left\lceil \frac{\|\mathcal{P}_j\|}{c_R} \right\rceil$. Specifically, when we move element $i$ from $\mathcal{P}_{j-1}$ to $\mathcal{P}_j$, $\left\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \right\rceil$ must decrease or remain the same. In the former ending condition (i.e., $\left\lceil \frac{\|\mathcal{P}_{j-1}\|}{c_R} \right\rceil < \left\lceil \frac{\|\mathcal{P}_j\|}{c_R} \right\rceil$), we swap elements according to the process mentioned in the proof of Theorem 3.3 so that the decreasing order of $\left\lceil \frac{\|\mathcal{P}_j\|}{c_R} \right\rceil$ is maintained and the elements in two partitions remains consecutive. In other words, we obtain two consecutive partitions $\mathcal{P}_{j-1}^{new}, \mathcal{P}_j^{new}$ and we have $\|\mathcal{P}_{j-1}^{new}\| = \|\mathcal{P}_j^{old}\|, \|\mathcal{P}_j^{new}\| = \|\mathcal{P}_{j-1}^{old}\|$. After we finish moving elements to $\mathcal{P}_j$ according to two ending conditions, we repeat the above process with $j = j - 1$ until $j = 1$ (i.e., until the size of the second largest partition is divisible by $c_R$). Proof completes. □

## D MICRO-BENCHMARK

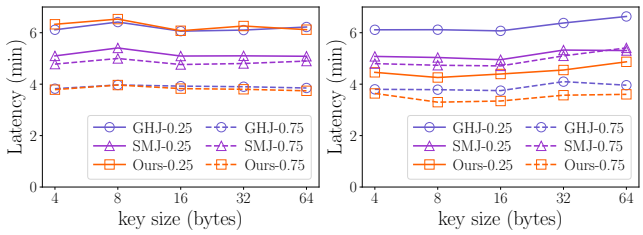
**Varying $k$.** In this experiment, we examine the effect of $k$ using 2 memory budget ($\sqrt{\|R\| \cdot F}/4$ and $3 \cdot \sqrt{\|R\| \cdot F}/4$ pages) by varying $k$ from $20K$ to $200K$. As mentioned earlier, tracking top-$k$ frequently matching keys does not necessarily mean we use top-$k$ elements to build the hash map/set in our approximation algorithm. As shown in Figure 11a, all the orange lines do not even change when $k$ grows. Larger $k$ does not yield significant performance improvement.




**(a) Uniform** **(b) Zipfian**

**Figure 11: Varying $k$.**

**Varying key size**. We further evaluate the impact of the key size since our approaches relies on the in-memory hash map to enforce the partitioning assignment. In this micro-benchmark. we use the same memory budget as before and vary the key size from 4 bytes to 64 bytes. Observe from Figure 12a and 12b, in most cases, the key size does not have huge impact over the join performance. Although larger key size indicates higher CPU complexity, this difference is negligible when the performance is dominated by #I/Os.




**(a) Uniform** **(b) Zipfian**

**Figure 12: Varying key size.**