NOCAP: Near-Optimal Correlation-Aware Partitioning Joins

ZICHEN ZHU, Boston University, USA
XIAO HU, University of Waterloo, Canada
MANOS ATHANASSOULIS, Boston University, USA

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. 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 state-of-the-art algorithms and mimics the best algorithm for a memory budget varying from below $\sqrt{\|\text{relation}\|}$ to more than $\|\text{relation}\|$.

CCS Concepts: • **Information systems** → **Join algorithms**; • **Theory of computation** → *Database query processing and optimization (theory).*

Additional Key Words and Phrases: Storage-based Join, Partitioning Join, Dynamic Hybrid Hash Join

ACM Reference Format:
Zichen Zhu, Xiao Hu, and Manos Athanassoulis. 2023. NOCAP: Near-Optimal Correlation-Aware Partitioning Joins. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 252 (December 2023), 27 pages. https://doi.org/10.1145/3626739

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, all the queries of industry-grade benchmarks like TPC-H [53] and most of Join Order Benchmark (JOB) [31] are PK-FK equi-joins. Recent research has focused on optimizing in-memory equi-joins [4–6, 9, 11, 34, 50, 55], however, as the memory prices scale slower than storage [36], the available memory might not always be sufficient to store both tables simultaneously, thus requiring a classical storage-based join [47]. This is common in a shared resource setting, like multiple colocated databases or virtual database instances deployed on the same physical cloud server [8, 13, 27]. Besides, in edge computing, memory is also limited, which is further exacerbated when other memory-demanding services are

Authors' addresses: Zichen Zhu, zczhu@bu.edu, Boston University, Boston, MA, USA; Xiao Hu, xiaohu@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Manos Athanassoulis, mathan@bu.edu, Boston University, Boston, MA, USA.

2836-6573/2023/12-ART252 $15.00
https://doi.org/10.1145/3626739

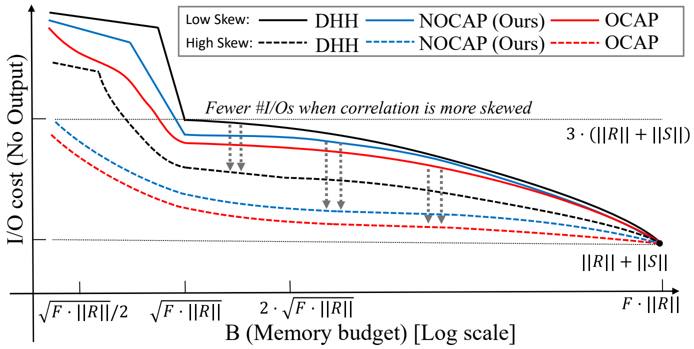


Fig. 1. NOCAP is closer to the optimal (OCAP) than the state of the art (DHH). Further, its benefits increase for higher skew in the join correlation. We assume $\|R\| \leq \|S\|$, and that F is the fudge factor indicating the space amplification factor between the in-memory hash table and the stored raw data.

running [24]. In several other data-intensive use cases like Internet-of-things, 5G communications, and autonomous vehicles [15, 21], memory might also be constrained. Finally, there are two main reasons we 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 resources must be shared among all the 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 [18, 30, 51], and is extensively used in existing database engines (e.g., MySQL [38], PostgreSQL [45], 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 [45] and AsterixDB [2, 28]) often implement a variant of HHJ, Dynamic Hybrid Hash join (DHH) that dynamically decides which partitions should stay in memory during partitioning.

The Challenge: Exploit Skew. For PK-FK joins, we describe the number of matching keys per PK using a *distribution*, which we also refer to as *join attribute correlation*, or simply *join correlation*. In turn, join correlation can be uniform (all PKs have the same number of FK matches) or skewed (some PKs have more matches than others). Although there are many skew optimizations for joins [19, 25, 29, 33, 39], 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. As such, while these heuristics may work well in some scenarios, they generally 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 [17]. 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 [46]). 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 primary relation and m is the memory budget in pages). 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. Further, we rely on OCAP to identify the headroom for improvement from the state of the art, and use it as a building block of a practical algorithm that we discuss next.

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* (NOCAP) algorithm. NOCAP enforces a strict memory budget and splits the available memory between buffering partitions and caching information regarding skew keys in join correlation. In Figure 1, we construct a conceptual graph to compare our solution with the state-of-the-art storage-based join method, Dynamic Hybrid Hash (DHH). As shown, DHH cannot fully adapt to the correlation and thus results in higher I/O cost than OCAP when the correlation is skewed. Our approximate algorithm, NOCAP, uses the same input from DHH and achieves near-optimal I/O cost, as observed when compared with OCAP. Further, while DHH is able to exploit the higher skew to reduce its I/O cost, the headroom for improvement for high skew is higher than for low skew, which is largely achieved by our approach. Overall, NOCAP is a practical join algorithm that is always 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, and propose an optimal correlation-aware partitioning (OCAP) algorithm based on dynamic programming (§3.1), assuming that the join correlation is known. The time complexity of OCAP with our pruning techniques is $O\left((n^2 \cdot \log m)/m\right)$ and the space complexity is $O(n)$, where n is the number of records in the primary relation and m is the memory budget in pages (§3.2).
- We design a near-optimal correlation-aware partitioning (NOCAP) algorithm based on OCAP. NOCAP uses partial correlation information (the same information used by the state-of-the-art skew-optimized join algorithms) and achieves near-optimal performance under memory budget constraints (§4). Our code is available at https://github.com/BU-DiSC/NOCAP-join.
- 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 [18, 23, 26, 47] (§2.1), and then present more details of Dynamic Hybrid Hash (DHH), a variant of the state-of-the-art HHJ (§2.2).

Table 1. Estimated Cost for NBJ, GHJ, and SMJ for $R \bowtie S$. $||R||$ and $||S||$ are the number of pages of R and S. Assume $||R|| \leq ||S||$. #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. #s-passes is the number of sorting passes of R and S needed to end up with fewer than $B-1$ sorted runs (B is the total number of available memory pages). μ and τ represent the write/read asymmetry [41], where RW, SW, and SR denote the latency per I/O for random write, sequential write, and sequential read, respectively.

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

2.1 Classical Storage-based Joins

Nested Block Join (NBJ). NBJ partially loads the smaller relation R (in *chunks* of size equal to the available memory) in the form of an in-memory hash table. For each chunk, the entire (larger) relation S is scanned once to produce the corresponding join output. 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 sorted runs is less than $B-1$, the last sorting phase can be combined with the multi-way join phase to avoid repetitive reads and writes [47].

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 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 memory budget 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 that stands for the space amplification factor between the in-memory hash table and the stored raw data), HHJ downgrades into GHJ because it is not feasible to maintain a partition in memory while having enough space in memory for the remaining partitions and the output.

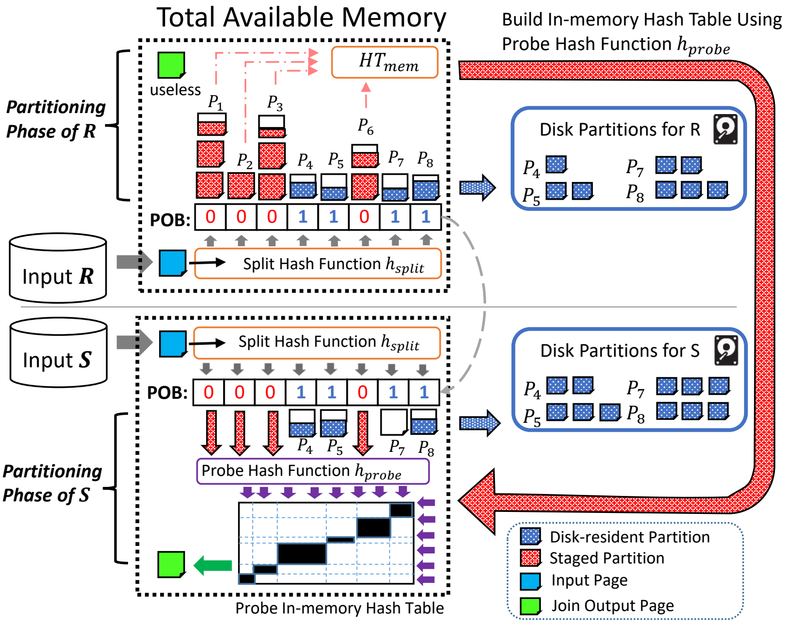


Fig. 2. An illustration of partitioning in DHH. Partitions in red are staged in memory when R is being partitioned, while the remaining ones in blue are written to disk.

2.2 Dynamic Hybrid Hash Join

Here, we review DHH, the state-of-the-art implementation of HHJ, and its skew optimization. While there are many different implementations of DHH [26], the core idea is the same across all: decide which partitions to stay in memory or on disk.

Framework. We visualize one implementation (AsterixDB [2, 28]) of DHH 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, thus freeing 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", short as POB, 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 is applied to build a large in-memory hash table using all staged partitions. It is worth noting that when no partitions can be staged in the memory (i.e., all the partitions are spilled out to disk), DHH naturally **downgrades** into GHJ. When partitioning relation S, DHH first applies the partitioning hash function and then checks the POB for every record. If it is not set, DHH directly probes the in-memory hash table and generates the join result. Otherwise, DHH moves the record to the output page of the corresponding partition and flushes this page to the disk if it is full. Finally, DHH performs exactly the same steps as traditional GHJ to join all the disk-resident partitions. The partitioning and probing of DHH is presented in detail in Algorithms 1 and 2.

Number of Partitions. The number of partitions in DHH (noted by m_{DHH}) is a key parameter that largely determines join performance. A large m_{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_{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

Algorithm 1: DHH-Primary(R, HS_{mem}, B)

```
$m_{\text{DHH}} \leftarrow \max\left\{20, \left\lceil \frac{\|R\|\cdot F - B}{B-1} \right\rceil\right\}$;
Initialize a length-$m_{\text{DHH}}$ array POB as $\{0, 0, ..., 0\}$;
Initialize a length-$m_{\text{DHH}}$ partitioning PartPages;
foreach $r \in R$ do
    PartID $\leftarrow h_{\text{split}}(r.\text{key})$;
    if POB[PartID] $= 0$ then
        if $\sum_{i=1}^{m_{\text{DHH}}}$ PartPages[$i$].size() $= B - 2 - m_{\text{DHH}}$ then
            Flush an arbitrary partition $j$ to disk;
            POB[$j$] $= 1$;
        else append $r$ to PartPages[PartID];
    else append $r$ to the in-disk partition PartID;
foreach $r \in$ PartPages do add $r$ to $HT_{\text{mem}}[h_{\text{probe}}(r.\text{key})]$;
return $HT_{\text{mem}}$, POB;
```

Algorithm 2: DHH-Foreign$(S, HT_{\text{mem}}, \text{POB})$

```
foreach $s \in S$ do
    if $h_{\text{probe}}(s.\text{key}) \in HT_{\text{mem}}$ then
        if $s$.key is in records $HT_{\text{mem}}[h_{\text{probe}}(s.\text{key})]$ then
            join $s$ with matched records and output;
    else
        if POB[$h_{\text{split}}(s.\text{key})$] $= 1$ then
            append $s$ to the in-disk partition PartID;
```

correlation skew, but past work on HHJ employs some heuristic rules. For example, an experimental study [26] recommends that 20 is the minimum number of partitions when 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_{DHH} as $m_{\text{DHH}} = \left\lceil \frac{\|R\|\cdot F - B}{B-1} \right\rceil$ [51], where $\|R\|$ is the size of relation R 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 $m_{\text{DHH}} = \max\left\{\left\lceil \frac{\|R\|\cdot F - B}{B-1} \right\rceil, 20\right\}$ (assuming $m_{\text{DHH}} \leq B - 2$ in most cases, otherwise we configure $m_{\text{DHH}} = B - 1$ and execute GHJ).

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 join correlation can be non-uniform. Histojoin [16] proposes to cache the Most Common Values (MCVs, maintained by DBMS) in memory to reduce the I/O cost. Specifically, if the memory budget is enough, Histojoin first partitions the dimension table R and caches entries with high-frequency keys in a small hash table in memory. When it partitions the fact table S, if any keys have a match in the small hash table, it directly joins them to produce the output. In this way, Histojoin avoids writing many records with the same foreign key from the fact table and reading them back into memory during the probing phase. However, it is unclear what degree of frequency should be treated as *high* to qualify for this optimization and how large the hash table for the skewed keys should be within the available memory budget. In one implementation of Histojoin [17], the memory allocated to the hash table specifically for skewed keys is limited to 2%. This restriction ensures that the specific hash table fits in memory. Likewise, in PostgreSQL [46], when the skew optimization knob is enabled, and MCVs are present in the system cache (thus not occupying the user-specified memory), PostgreSQL follows a similar approach by adding an extra trigger to build the hash table for skewed keys. By default, PostgreSQL also allocates a fixed budget of 2% memory for the hash table dedicated to handling skewed keys, but this allocation only occurs when the total frequency of skewed keys is more than $0.01 \cdot n_S$ (n_S is the number of records in relation S). In fact, PostgreSQL implements a general version of Histojoin by specifying the frequency threshold. However, existing skew optimizations in both PostgreSQL and Histojoin rely on fixed thresholds (i.e., memory budget for skewed keys, and frequency threshold), which apparently cannot adapt to different memory budget and different join skew. For example, as shown in Figure 1, state-of-the-art DHH is not enough to achieve the ideal I/O cost and is further away from the optimal for higher skew.

3 OPTIMAL CORRELATION-AWARE PARTITIONING JOIN

We now discuss the theoretical limit of an I/O-optimal partitioning-based join algorithm when the exact distribution of the join correlation is known in advance and can be accessed for free. We assume that R is the relation that holds the primary key (dimension table) and S is the relation that holds the foreign key (fact table). More specifically, we model the correlation between two input relations with respect to the join attributes as a *correlation table* CT, where $\mathsf{CT}[i]$ is the number of records in S matching the i-th record in R. Our goal is to find which keys should be cached in memory, and how the rest of the keys should be partitioned in order to minimize the total I/Os of the join execution with an arbitrary memory budget.

We start with an easy case when no records can be cached during the partition phase (§3.1) and 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 a theoretically I/O-optimal algorithm, which helps us understand the lower bound of any feasible partitioning but cannot be converted into a practical algorithm. All missing proofs and details are given in our full version [58].

3.1 Optimal Partitioning Without Caching

We first build the cost model for partitioning assuming no cached records during the partitioning (§3.1.1), then present the main theorem for characterizing the optimal partitioning (§3.1.2) and find the optimal partitioning efficiently via dynamic programming (§3.1.3).

3.1.1 Partitioning as An Integer Program. A partitioning $\mathbb{P}$ assigns n records from R to m partitions. We use one input page and the rest of the pages as output buffers for each partition, thus, $m \leq B-1$. More specific constraints on m will be determined in §3.2 and §4.1. Once we know how to partition R (the primary key relation), we can apply the same partition strategy to S.

Partitioning. We encode a partitioning $\mathbb{P}$ as a Boolean matrix of size $n \times m$, where $\mathbb{P}_{i,j} = 1$ indicates that the i-th record belongs to the j-th partition. If the i-th record from R does not have a match in S, i.e., $\mathsf{CT}[i] = 0$, we will not assign it to any partition. We can preprocess the input to filter out these records so that the remaining records belong to exactly one partition. In this way, a partitioning $\mathbb{P}$ can be compressed as a mapping function $f : [n] \rightarrow [m]$ from the index of a record to its partition index in $\mathbb{P}$, such that the i-th record from relation R is assigned to the $f(i)$-th partition in $\mathbb{P}$. Let $P_j = \{i \in [n] : f(i) = j\}$ be the set of indexes of records from R assigned to the j-th partition in $\mathbb{P}$. We can then derive the number of pages for R_j and S_j as $\lVert R_j \rVert = \lceil |P_j|/b_R \rceil$, $\lVert S_j \rVert = \lceil \sum_{i \in P_j} \mathsf{CT}[i]/b_S \rceil$, where b_R (resp. b_S) is the number of records per page for R (resp. S). Moreover, we use $\mathcal{N}_f(i) = \{i' \in [n] : f(i') = f(i)\}$ to denote the records from S that fall into the same partition with the i-th record in R. We summarize our notation in Table 2.

Table 2. Summary of our notation.

Notation	Definitions (Explanations)
$\lVert R \rVert$ ($\lVert S \rVert$)	#pages of relation R (S)
n_R (n_S)	#records from relation R (S)
P_j	the j-th partition ($P_j = R_j$)
$\lvert P_j \rvert$	#unique keys that belong to the j-th partition
$\lVert R_j \rVert$ ($\lVert S_j \rVert$)	#pages of a partition R_j (S_j)
$\mathbb{P}$	a Boolean matrix for partitioning assignation
$f : i \longrightarrow j$	returns j for i so that $\mathbb{P}_{i,j} = 1$
$\mathcal{N}_f(i)$	the partition where the i-th record belongs
n	$n = n_R$ in the partitioning context
m	#partitions
B	#pages of the total available buffer
F	the fudge factor of the hash table
CT	the correlation table of keys and their frequency
$\mathsf{CT}[a : b]$	a sub-array of CT which starts at index a and ends at b (inclusive)
b_R (b_S)	#records from relation R (S) per page
c_R (c_S)	#records from relation R (S) per chunk in NBJ

Cost Function. Given a partitioning $\mathbb{P}$, the next step is to pair-wise join the partitions of S and R. Reusing the cost model in Table 1, the cost of computing the join results for a pair of partitions (R_j, S_j) is $\min\{\mathsf{NBJ}(R_j, S_j), \mathsf{SMJ}(R_j, S_j), \mathsf{GHJ}(R_j, S_j)\}$, by picking the cheapest one among NBJ, SMJ, and GHJ. We omit DHH here for simplicity, which does not lead to major changes in the optimal partitioning since, in most cases, NBJ is the most efficient one when taking read/write asymmetry into consideration (writes are slower than reads for modern SSDs [41, 42]). For ease of illustration, we assume only NBJ is always applied in the pair-wise join, i.e., the *join cost* induced by partitioning $\mathbb{P}$ is $\mathsf{Join}(\mathbb{P}, m) = \sum_{j=1}^{m} \mathsf{NBJ}(R_j, S_j)$.

Recall the cost function of NBJ in Table 1. For simplicity, we assume $\lVert R_j \rVert \leq \lVert S_j \rVert$ for each $j \in [m]$. The other case with $\lVert R_j \rVert \geq \lVert S_j \rVert$ can be discussed similarly. By setting #chunks= $\lceil \lVert R_j \rVert / ((B-2)/F) \rceil$, $\lVert R_j \rVert = |P_j|/b_R$ and $\lVert S_j \rVert = \lceil \sum_{i \in P_j} \mathsf{CT}[i]/b_S \rceil$, we obtain the cost function as follows:

$$\begin{aligned}\mathsf{Join}(\mathbb{P}, m) = \lVert R \rVert + \sum_{j=1}^{m} \left\lceil \frac{|P_j|}{c_R} \right\rceil \cdot \left\lceil \sum_{i \in P_j} \frac{\mathsf{CT}[i]}{b_S} \right\rceil &\leq \lVert R \rVert + m + \frac{n_R}{c_R} + \sum_{j=1}^{m} \left\lceil \frac{|P_j|}{c_R} \right\rceil \cdot \sum_{i \in P_j} \frac{\mathsf{CT}[i]}{b_S} \\ &= \lVert R \rVert + m + \frac{n_R}{c_R} + \frac{1}{b_S} \cdot \sum_{i=1}^{n} \mathsf{CT}[i] \cdot \left\lceil \frac{|\mathcal{N}_f(i)|}{c_R} \right\rceil,\end{aligned}$$

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 of reading and writing both relations during the partitioning phase is the same for all the partitioning strategies.

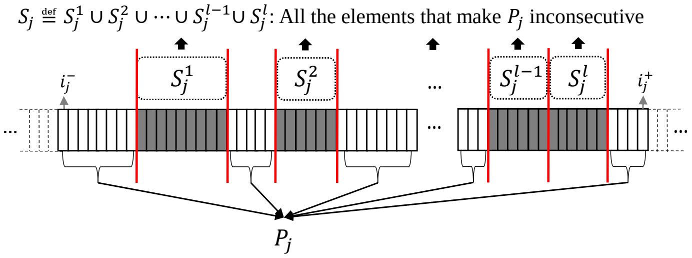

$S_j \overset{\text{def}}{=} S_j^1 \cup S_j^2 \cup \cdots \cup S_j^{l-1} \cup S_j^l$: All the elements that make P_j inconsecutive
S_j^1
S_j^2
...
S_j^{l-1}
S_j^l
i_j^-
i_j^+
...
P_j

Fig. 3. An illustration of records from a single partition, P_j, positioned in a non-consecutive way w.r.t. CT.

Integer Program. Given the number of partitions m, finding a partitioning $\mathbb{P}$ for R (containing the primary key) of n records with minimum cost can be captured by the following integer program:

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

However, solving this program needs to exhaustively search all possible partitioning strategies, thus having an exponential cost. We now characterize three key properties of the optimal partitioning, which allows us to use dynamic programming as well as further pruning of the search space.

3.1.2 Main Theorem. For simplicity, we assume that CT is *sorted in ascending order*, i.e., if $i_1 \leq i_2$, then $\mathsf{CT}[i_1] \leq \mathsf{CT}[i_2]$. With sorted CT, we present our main theorem for characterizing the optimal partitioning $\mathbb{P}$ with three critical properties. First, each partition of $\mathbb{P}$ contains consecutive records from the sorted CT array. Second, the sizes of all partitions are in a weakly descending order. Third, the size of each partition – except the first one in a weakly descending order – is divisible by c_R.

Theorem 3.1 (Main Theorem). *Given an arbitrary sorted* CT *array, there is an optimal partitioning* $\mathbb{P} = \langle P_1, P_2, \cdots, P_m \rangle$ *such that*

- **(consecutive property)** *for any* $i_1 \leq i_2$, *if* $i_1 \in P_j$ *and* $i_2 \in P_j$, *then* $i \in P_j$ *holds for any* $i \in [i_1, i_2]$;
- **(weakly-ordered property)** $\left\lceil \frac{|P_1|}{c_R} \right\rceil \geq \left\lceil \frac{|P_2|}{c_R} \right\rceil \geq \cdots \geq \left\lceil \frac{|P_m|}{c_R} \right\rceil$;
- **(divisible property)** *for any* $2 \leq i \leq m$, $|P_i|$ *is divisible by* c_R.

where P_j *is the set of indexes of records from* R *assigned to the* j*-th partition, i.e.,* $P_j = \{i \in [n] : f(i) = j\}$.

Lemma 3.2 (Swap Lemma). *For a partitioning* $\mathbb{P}$, *if there exists a pair of indexes* i_1, i_2 *with* $\mathsf{CT}[i_1] > \mathsf{CT}[i_2]$ *and* $|\mathcal{N}_f(i_1)| \geq |\mathcal{N}_f(i_2)|$, *it is feasible to find a new partitioning* $\mathbb{P}'$ *by swapping the partition assignment of records at* i_1 *and* i_2, *such that* $\mathsf{Join}(\mathbb{P}) \geq \mathsf{Join}(\mathbb{P}')$.

To prove the main theorem, the high-level idea is to transform an arbitrary partitioning $\mathbb{P}$ into $\mathbb{P}'$ with desired properties, while keeping the join cost of $\mathbb{P}'$ always smaller (or at least not larger) than that of $\mathbb{P}$. The core idea is a *swap* procedure, as described by Lemma 3.2, which can reassign records in a way that records with larger CT values are moved to a smaller partition. The proof of Lemma 3.2 directly follows the definition of join cost. Next, we will show step by step how to transform an arbitrary partitioning $\mathbb{P}$ into the desired $\mathbb{P}'$.

Algorithm 3: SWAP($\mathbb{P}$)

```
while there exists $P_j \in \mathbb{P}$ that is non-consecutive do
    while true do
        left, right $\leftarrow \min_{i\in[n]:f(i)=j} i, \max_{i\in[n]:f(i)=j} i$;
        $S_j \leftarrow \{i \in [n] : \text{left} \le i \le \text{right}, f(i) \neq j\}$;
        if $S_j = \emptyset$ then break;
        $\ell \leftarrow \min_{i\in S_j} i$;
        if $|\mathcal{N}_f(\ell)| > |P_j|$ then
            SWAP$(f(\ell), f(\text{left}))$, left $\leftarrow$ left + 1;
        else SWAP$(f(\ell), f(\text{right}))$, right $\leftarrow$ right $-$ 1;
return $\mathbb{P}$;
```

Consecutive Property. Given an arbitrary partitioning $\mathbb{P}$, we first show how to transform it into $\mathbb{P}'$ such that $\mathbb{P}'$ satisfies the consecutive property while $\text{Join}(\mathbb{P}') \le \text{Join}(\mathbb{P})$. As described in Algorithm 3, we first identify a partition whose records are not consecutive on the sorted CT, say P_j, and apply the swap procedure to it. Let i_j^- and i_j^+ be the minimum and maximum index of records from P_j on CT. For all records that make P_j non-consecutive, we can use multiple segments to cover them. The segments are defined as follows. 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 $j' \neq j$ ($S_j^{\ell_1}$ 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 the records that make P_j non-consecutive with respect to the sorted CT (Figure 3).

Algorithm 3 maintains two pointers left and right to indicate the smallest and largest indexes of records to be swapped. Consider the smallest index $\ell \in S_j$. If $|\mathcal{N}_f(\ell)| > |P_j|$, we swap ℓ and left by putting the record at position ℓ into $P_{f(\text{left})}$ and the record at position left into $P_{f(\ell)}$ ($P_{f(\ell)} = \mathcal{N}_f(\ell)$ by definition), noted by SWAP$(f(\ell), f(\text{left}))$. Otherwise, we swap ℓ and right similarly. We repeat the above procedure until all the partitions become consecutive on the sorted CT. The resulting partitioning from SWAP($\mathbb{P}$) is *consecutive* in terms of its CT positions, and the join cost does not increase as a part of this process, as shown by the swap lemma.

Algorithm 4: ORDER($\mathbb{P}$)

```
while there exists $P_{j'} \lesssim P_j \in \mathbb{P}$ s.t. $\left\lceil \frac{|P_{j'}|}{c_R} \right\rceil < \left\lceil \frac{|P_j|}{c_R} \right\rceil$ do
    $\ell, \ell' \leftarrow \min_{i\in P_j} i, \min_{i\in P_{j'}} i$;
    for $i \leftarrow 0$ to $|P_{j'}|$ do SWAP$(f(\ell + i), f(\ell' + i))$;
    $\mathbb{P} \leftarrow$ SWAP($\mathbb{P}$);
```

Weakly-Ordered Property. Given a partitioning $\mathbb{P}$ with the consecutive property, we next show how to transform it into $\mathbb{P}'$ such that $\mathbb{P}'$ also satisfies the weakly-ordered property. Since each partition in $\mathbb{P}$ contains consecutive records w.r.t. CT, we define a partial ordering $\lesssim$ on the partitions in $\mathbb{P}$. For simplicity, we assume that $P_1 \lesssim P_2 \lesssim \cdots \lesssim P_m$. As described in Algorithm 4, we always identify a pair of partitions that are **not** *weakly-ordered*, i.e., $P_{j'} \lesssim P_j$ but $\lceil |P_{j'}|/c_R \rceil < \lceil |P_j|/c_R \rceil$ (if such partitions exist). In this case, we apply Lemma 3.2 to swap records between $P_{j'}$ and P_j. This swap would turn P_j into non-consecutive on the sorted CT. If this happens, we further invoke Algorithm 3 to transform P_j into a consecutive one; then, $P_{j'}$ will be located after P_j in the sorted

CT. We will repeatedly apply the above procedure until there is no unordered pair of partitions in $\mathbb{P}$. We illustrate the weakly-ordered property, i.e., that the number of passes per partition decreases as the corresponding CT values increase, in Figure 4. In fact, a strongly-ordered property also holds by replacing $\lceil |P_j|/c_R \rceil$ with $|P_j|$. We keep the weakly-ordered version here to avoid conflict with the divisible property discussed below.

Divisible Property. Given a partitioning $\mathbb{P}$ satisfying the consecutive and weakly-ordered property, we next show how to transform it into $\mathbb{P}'$ such that $\mathbb{P}'$ also satisfies the divisible property. We start checking the m-th partition P_m. If $|P_m|$ is divisible by c_R, we skip it. Otherwise, we move records from the previous partition P_{m-1} to P_m until $|P_m|$ is divisible by c_R. If, after moving, the weakly-ordered property does not hold (i.e., $\lceil |P_{m-1}|/c_R \rceil < \lceil |P_m|/c_R \rceil$), we execute Algorithm 4 to maintain the weakly-ordered property. After ensuring $|P_m|$ is divisible, we move to P_{m-1}. We continue doing this until $|P_m|, \cdots, |P_2|$ are all divisible by c_R. This fine-grained movement between adjacent partitions naturally preserves the consecutive property and does not increase the join cost.

3.1.3 Dynamic Programming. With Theorem 3.1, we can now reduce the complexity of finding the optimal partitioning by resorting to dynamic programming (instead of a brute-force approach), searching candidate partitionings with consecutive, weakly-ordered, and divisible properties.

Formulation by Consecutive Property. We define a sub-problem parameterized by (i, j): finding the optimal partitioning for the first i records in the sorted CT using j partitions. The solution of this sub-problem parameterized by (i, j) is denoted as $V[i][j]$. Recall that the optimal solution of our integer program in Section 3.1.1 is exactly $V[n][m]$. As described by Algorithm 5, we can compute V in a bottom-up fashion. Specifically, we use $V[i][j]$.cost to store the cost of an optimal partitioning for the sub-problem parameterized by (i, j) and use $V[i][j]$.LastPar to store the starting position of the last partition. We also define CalCost(s, e) as the per-partition join cost PPJ$(R_{j'}, S_{j'})$, where $s(e)$ is the position of the first (last) record in partition j' with respect to CT, in other words $P_{j'} = \{i | i \in [s, e]\}$. Note that we omit the term $\|R_{j'}\|$ from NBJ$(R_{j'}, S_{j'})$ because it will always sum up to $\|R\|$. Now, CalCost(s, e) can be easily given using the pre-computed prefix sum:

$$\text{CalCost}(s, e) = \left(\sum_{i=1}^{e} \text{CT}[i] - \sum_{i=1}^{s-1} \text{CT}[i] \right) \cdot \left\lceil \frac{e - s + 1}{c_R} \right\rceil \quad (1)$$

The core idea of Algorithm 5 is the following recurrence formula:

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

which iteratively searches for 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 from each index to its associated partition, as described in Algorithm 6.

The time complexity of Algorithm 5 is $O(m \cdot n^2)$, which is dominated by three for-loops, and the space complexity is $O(m \cdot n)$ for storing V. This significantly improves the brute-force enumerating method, whose time complexity is as large as $O(m^n)$.

Speedup by Weakly-Ordered Property. Combing the weakly-ordered property in Theorem 3.1 with the pigeonhole principle [20], for any sub-problem parameterized by (i, j), there exists an optimal partitioning such that: (i) the *"largest"* partition contains at least the first $\left\lfloor \frac{i-1}{j} \right\rfloor + 1 - c_R$ records in the sorted CT, and (ii) the *"smallest"* partition contains at most the last $\left\lfloor \frac{i}{j} \right\rfloor + c_R$ records in the sorted CT. Note that the *"largest"* (*"smallest"*) partition is not always the largest (smallest) in terms of the actual partition size. It is possible that $P_{j'} \precsim P_{j'+1}$ and $\lceil |P_{j'}|/c_R \rceil = \lceil |P_{j+1}|/c_R \rceil$ but $|P_{j'}| \le |P_{j'+1}|$ (we must have $|P_{j'}| + c_R \ge |P_{j'+1}|$ in this case). As such, the *"largest"* partition for

Algorithm 5: PARTITION(CT, n, m)

```
1  Initialize $V$ of sizes $n \times m$;
2  for $i \leftarrow 1$ to $n$ do
3      $V[i][1]$.cost ← CalCost$(1, i)$,  $V[i][1]$.LastPar ← 1;
4  for  $i \leftarrow 2$ to $n$ do
5      for  $j \leftarrow 2$ to $\min(m, i)$ do
6          $V[i][j]$.cost ← $+\infty$;
7          for  $k \leftarrow 0$ to $i - 1$ do
8              tmp ← $V[k][j-1]$.cost + CalCost$(k+1, i)$;
9              if tmp < $V[i][j]$.cost then
10                 $V[i][j]$.cost ← tmp;
11                 $V[i][j]$.LastPar ← $k + 1$;
12 return $V$;
```

Algorithm 6: GETCUT(V, n, m)

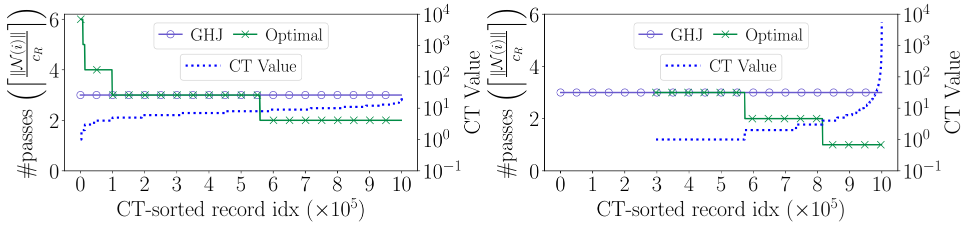
```
LastPar ← $n$;
for $j \leftarrow 0$ to $m - 1$ do
    for $i \leftarrow V$[LastPar][$m - j$].LastPar to LastPar do $f(i) \leftarrow m - j$;
    LastPar ← $V$[LastPar][$m - j$].LastPar − 1;
return $F$;
```

the sub-problem parameterized by (i, j) cannot be smaller than $\left\lfloor \frac{i-1}{j} \right\rfloor + 1 - c_R$. A similar analysis can be applied to the upper bound of the size of the *"smallest"* partition. Synthesizing the above analysis with the weakly-ordered property and the divisible property (see more details in our full version [58]), we get tighter bounds of k in line 7 of Algorithm 5:

- $k \geq \max\left\{(i - c_R) \cdot \left(1 - \frac{1}{j}\right), 0\right\}$**.** Partition P_j contains the records in positions $[k+1, i]$, and it should not be larger than the last (*"smallest"*) partition in $V[k][j-1]$. The last (*"smallest"*) partition for the sub-problem parameterized by $(k, j-1)$ is at most $\left\lfloor \frac{k}{j-1} \right\rfloor + c_R$. Hence, $i - k \leq \left\lfloor \frac{k}{j-1} \right\rfloor + c_R$.
- $k \leq \min\left\{i - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1 + c_R, i - 1\right\}$**.** Partition P_j contains the records in positions $[k+1, i]$, and it should not be smaller than the *"largest"* partition of remaining records in positions $[i+1, n]$. Recall that we still need to put the remaining $n - i$ records into $m - j$ partitions. The *"largest"* partition of the sub-problem parameterized by $(n - i, m - j)$ contains at least $\left\lfloor \frac{n-i-1}{m-j} \right\rfloor + 1 - c_R$ records (here this sub-problem is defined on the last $n - i$ records). Hence, $i - k \geq \left\lfloor \frac{n-i-1}{m-j} \right\rfloor + 1 - c_R$.

After changing the range of the third loop (line 7, Algorithm 5) to:

$$\left[\max\left\{(i - c_R) \cdot \left(1 - \frac{1}{j}\right), 0\right\}, \min\left\{i - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1 + c_R, i - 1\right\}\right], \tag{2}$$



(a) Uniform (b) Zipfian

Fig. 4. A comparison of the number of passes required for the outer relation S between the partitioning adopted by DHH/GHJ and our optimal partitioning, assuming $B \leq \sqrt{\|R\| \cdot F}$. Note that DHH without skew optimization downgrades to GHJ. We assume $n_R = 1M$, $n_S = 8M$, $F = 1.02$, and $B = 320$ pages. Moreover, the record size of R is $1KB$, thus R occupies $250K$ pages. (a) For random (uniform) partitioning (GHJ), each partition is around $250K/319 \approx 784$ pages, which requires $\lceil 784/(318 \times F) \rceil = 3$ passes to scan every partition of S. In contrast, the optimal partitioning allows the number of passes to vary from 2 to 6, which follows a non-monotonically decreasing trend as the CT value increases. (b) For a skewed join correlation, we observe a similar pattern. Further, the optimal partitioning naturally excludes records with $\mathsf{CT}[i] = 0$.

we can skip the calculation of $V[i][j]$ when $(i - c_R) \cdot \left(1 - \frac{1}{j}\right) > i - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1 + c_R$. This pruning reduces the time complexity to $O(n^2 \log m)$.

Speedup by Divisible Property. To utilize the divisible property, we initially put the first (n mod c_R) records in the first partition and then change the step size (lines 4 and 7) of Algorithm 5 to c_R. This way, we can shrink V to a $(\lceil n/c_R \rceil + 1) \times m$ matrix. 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 is reduced to $O\left((n/c_R)^2 \log m\right) = O\left(\frac{n^2 \log m}{m^2}\right)$ and the space complexity to $O\left(n/c_R \cdot m\right) = O(n)$.

3.2 Optimal Partitioning With Caching

We move to the general case when records can be cached. In this case, some records stay in memory during the partitioning phase to avoid repetitive reads/writes, while the remaining records still go through the partitioning phase as Section 3.1. A natural question arises: *which records should stay in memory and which should be written to disk?* As we can see in DHH (Figure 2), the hash function h_{split}, together with POB categorizes all the join keys into two sets, memory-resident keys as $K_{\text{mem}} \subseteq [n]$ and disk-resident keys as $K_{\text{disk}} = [n] - K_{\text{mem}}$. Then, it suffices to find an optimal partitioning (without caching) for K_{disk}, which reduces to our problem in Section 3.1. Taking into account the partitioning cost (that includes writing data back to disk), we derive the general join cost function $\mathsf{Join}(K_{\text{disk}}, \mathbb{P}, m)$ as:

$$\mathsf{Join}(K_{\text{disk}}, \mathbb{P}, m) = \left\lceil \sum_{i \in K_{\text{disk}}} \frac{\mathsf{CT}[i]}{b_S} \cdot \left\lceil \frac{|\mathcal{N}_f(i)|}{c_R} \right\rceil \right\rceil + (1+\mu) \cdot \left\lceil \frac{|K_{\text{disk}}|}{b_R} \right\rceil + \mu \cdot \sum_{i \in K_{\text{disk}}} \left\lceil \frac{\mathsf{CT}[i]}{b_S} \right\rceil,$$

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

Algorithm 7: OCAP(CT, n, B)

```
tmp ← $+\infty$, $f \leftarrow \emptyset$;
for $k \leftarrow 0$ to $c_R$ do
    $V \leftarrow$ PARTITION($\text{CT}[1:n-k], n-k, B-2-\lceil \frac{k \cdot F}{b_R} \rceil$);
    $c_{\text{probe}} \leftarrow \lceil \frac{n-k}{b_R} \rceil + V[n-k][B-2-\lceil \frac{k \cdot F}{b_R} \rceil]$.cost;
    $c_{\text{part}} \leftarrow \mu \cdot (\lceil \frac{n-k}{b_R} \rceil + \lceil \frac{1}{b_S} \cdot \sum_{i=1}^{n-k} \text{CT}[i] \rceil)$;
    if tmp $< c_{\text{probe}} + c_{\text{part}}$ then
        tmp $\leftarrow c_{\text{probe}} + c_{\text{part}}$;
        $f \leftarrow$ GETCUT($V, n-k, B-2-\lceil \frac{k \cdot F}{b_R} \rceil$);
return $f$;
```

R in the probing phase. Putting everything together, we have the following integer program:

$$\begin{aligned} \min_{K_{\text{disk}}, \mathbb{P}} \quad & \text{Join}(K_{\text{disk}}, \mathbb{P}, m) \\ \text{subject to} \quad & \left\lceil \frac{n - |K_{\text{disk}}|}{b_R} \cdot F \right\rceil + m + 2 \leq B, \\ & \sum_{j=1}^{m} \mathbb{P}_{i,j} = 1, \forall i \in [n] \\ & \mathbb{P}_{i,j} \in \{0,1\}, \forall i \in [n], j \in [m], \end{aligned}$$

where the first constraint limits the size of the memory-resident hash table, as two pages are reserved for input and join output, and m pages for the output buffer for each disk-resident partition.

The integer program above can be interpreted by first fixing the K_{disk} (i.e., which keys/records should be spilled out to disk), and then finding the optimal partitioning for K_{disk}. Algorithm 5 can be invoked as a sub-routine for any given K_{disk}. We still need to identify the optimal K_{disk} (i.e., which records should be spilled out to disk) that leads to the overall minimum cost. Although there is an exponentially large number of candidates for K_{disk}, it is easy to see that records with low CT values should be spilled out to disk, while records with high CT values should be kept in memory. This follows from the fact that the new objective function only uses the CT values of K_{disk}. In other words, the consecutive property and the weakly-ordered property of Theorem 3.1 still apply by treating the cached keys as a special partition P_{m+1}, and treating $|P_{m+1}|$ as 0 in the weakly-ordered property (since P_{m+1} does not yield any extra I/Os). We can achieve the optimal cost by caching the top-k records from CT in memory, if restricting the size of K_{mem} to be k, where $k < c_R$ since at most c_R records can remain in memory. By extending Algorithm 5 to allow records to be cached in memory during partitioning, we finalize OCAP in Algorithm 7.

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 5 at most c_R times. Recall that the complexity of Algorithm 5 can be reduced to $O\left(\left(n^2 \log m\right)/c_R^2\right)$, thus, the time complexity for OCAP is now $O\left(\left(n^2 \log m\right)/c_R\right) = O\left(\left(n^2 \log m\right)/m\right)$. The space complexity remains $O(n)$.

4 OUR PRACTICAL ALGORITHM

In practice, it is impossible to know the exact correlation between input relations for each key in advance; instead, a much smaller set of high-frequency keys are collected for skew optimization, such as the Most Common Values (MCVs) in PostgreSQL [46]. PostgreSQL supports skew optimization

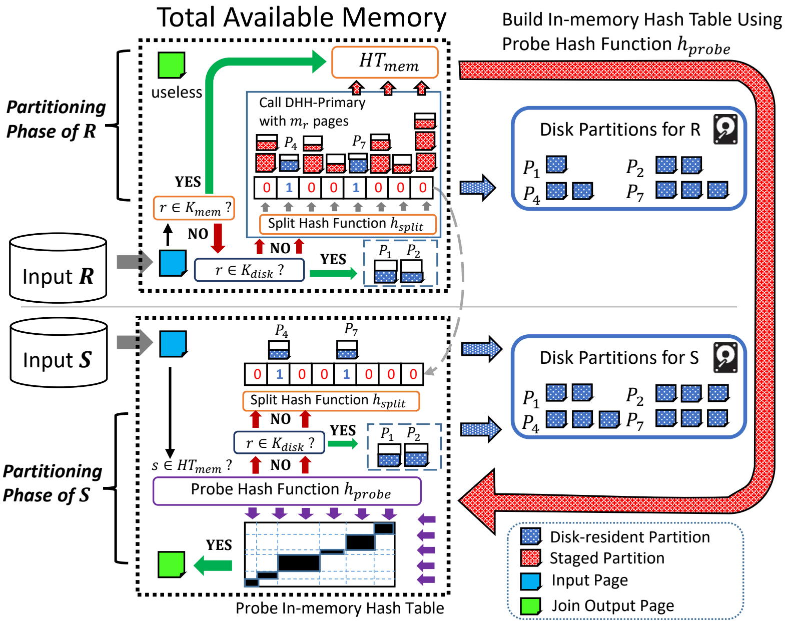


Fig. 5. An illustration of the hybrid partitioning workflow of NOCAP. Partitions P_4 and P_7 are spilled to disk when invoking DHH, while P_1 and P_2 are written to disk due to the partitioning specified by f_{disk}. Other records in R either have keys from K_{mem} or are staged in memory when invoking DHH, which will be inserted to the in-memory hash table HT_{mem}.

by assigning 2% memory to build a hash table for skewed keys if their total frequency is larger than 1% of the outer relation, but this heuristic uses fixed thresholds to trigger skew optimization, which may be suboptimal for an arbitrary join correlation. Hence, we present a practical algorithm built upon the theoretically I/O-optimal algorithm in the previous section. Instead of relying on the whole correlation CT table, our practical algorithm only needs the same amount of MCVs as PostgreSQL and generates a hybrid partitioning that can adapt to an arbitrary join correlation and constrained memory budget.

4.1 Hybrid Partitioning

Let K be the set of skew keys tracked from MCVs with their CT values known. Guided by the consecutive and weakly-ordered properties in Theorem 3.1, for a given $|K_{mem}|$, it is always preferable to keep in memory the keys with the highest CT values. That way, we minimize the number of records we partition to disk, thus minimizing disk accesses. For the rest of the keys that are not tracked by MCVs, we consider their frequency as low and can be handled efficiently by traditional hash joins, GHJ or DHH.

Framework. In a hybrid partitioning, we design a hash set, HS_{mem}, to store a subset of the skewed keys (denoted as $K_{\text{mem}} \subseteq K$) that will be used to build the in-memory hash table, HT_{mem}, a hash map, f_{disk}, to store another subset of skewed keys (denoted as $K_{\text{disk}} \subseteq K$) with their designated partition identifiers, and the rest of the skewed keys (if any) together with non-skewed keys (denoted as K_{rest}) will be uniformly partitioned by GHJ or DHH with a certain memory budget m_{rest} (in pages). With HS_{mem}, f_{disk} and m_{rest} in hand, we can partition the records of R using Algorithm 8. For each record in R, if its key is in HS_{mem}, we put the entire record in HT_{mem} (HS_{mem} only stores the key); if its key belongs to f_{disk}, we put it into the partition specified by f_{disk}, which will be spilled to disk

Algorithm 8: Hybrid-Partition-Primary$(R, HS_{\text{mem}}, f_{\text{disk}}, m_{\text{rest}})$

```
$R' \leftarrow \{r \in R : r.\text{key} \notin HS_{\text{mem}} \wedge r.\text{key} \notin f_{\text{disk}}\}$; ▷ Logically construct $R'$ without instantiation;
foreach $r \in R - R'$ do
    if $r.\text{key} \in HS_{\text{mem}}$ then add $r$ to $HT_{\text{mem}}[h_{\text{probe}}(r.\text{key})]$;
    else assign $r$ to disk-resident partition $f_{\text{disk}}(r.\text{key})$;
$(HT'_{\text{mem}}, \text{POB}) \leftarrow$ DHH-Primary$(R', m_{\text{rest}})$;
return $(HT'_{\text{mem}} \cup HT_{\text{mem}}, \text{POB})$
```

Algorithm 9: Hybrid-Partition-Foreign$(S, HT_{\text{mem}}, f_{\text{disk}}, \text{POB})$

```
$S' \leftarrow \{s \in S : s.\text{key} \in f_{\text{disk}}\}$;            ▷ Logically construct $S'$ without instantiation;
foreach $s \in S'$ do
    assign $s$ to disk-resident partition $f_{\text{disk}}(s.\text{key})$
DHH-Foreign$(S - S', HT_{\text{mem}}, \text{POB})$;
```

Algorithm 10: NOCAP(CT, k, n, m)

```
$c_{opt} \leftarrow +\infty$, $k_{\text{mem}} \leftarrow 0$, $k_{\text{disk}} \leftarrow 0$, $f_{\text{disk}} \leftarrow \emptyset$;
for $i_1 \leftarrow 0$ to $\min(k, c_R)$ do
    $V \leftarrow$ Partition $\left(\text{CT}[i_1 + 1 : k], k - i_1, \left\lceil \frac{k - i_1}{c_R} \right\rceil\right)$;
    for $i_2 \leftarrow 0$ to $\min(k, c_R) - i_1$ do
        for $j \leftarrow \min(i_2, 1)$ to $\lceil \frac{i_2}{c_R} \rceil$ do
            $m_{\text{rest}} \leftarrow B - 2 - B_{HT}(i_1) - B_{HS}(i_1) - B_f(i_2) - j$;
            $c_{\text{probe}} \leftarrow V[i_2][j].\text{cost}$;
            $c_{\text{part}} \leftarrow \mu \cdot \left(\left\lceil \frac{i_2}{b_R} \right\rceil + \left\lceil \frac{1}{b_S} \cdot \sum_{j=i_1+1}^{j=i_2} \text{CT}[j] \right\rceil\right)$;
            $c_{\text{rest}} \leftarrow g_{\text{DHH}}(i_2 + 1, n, m_{\text{rest}})$;
            $c_{\text{tmp}} \leftarrow c_{\text{probe}} + c_{\text{part}} + c_{\text{rest}}$;
            if $c_{\text{tmp}} < c_{\text{opt}}$ then
                $c_{opt} \leftarrow c_{\text{tmp}}$, $k_{\text{mem}} \leftarrow i_1$, $k_{\text{disk}} \leftarrow i_2$;
                $f_{\text{disk}} \leftarrow$ GetCut$(V, i_2, j)$;
return $k_{\text{mem}}, k_{\text{disk}}, f_{\text{disk}}$;
```

later; otherwise, it will be handled by GHJ or DHH along with the rest of the records using the m_{rest} pages. We can similarly partition records in S using Algorithm 9. This hybrid partitioning workflow is visualized in Figure 5.

We note the following difference regarding whether to use GHJ or DHH. When the available memory is large, say $\lceil |K_{\text{rest}}|/c_R \rceil < m_{\text{rest}}$, we simply invoke DHH to partition K_{rest} using m_{rest} pages. When the available memory is small, as discussed in Section 2.2, DHH downgrades to GHJ.

Enforcing Memory Constraints. State-of-the-art systems (e.g., MySQL and PostgreSQL) typically assign to each join operator a user-defined memory budget. The default memory limit for the join operator in MySQL is 256 KB [37], and the one in PostgreSQL is 4 MB [44]. Assuming the available

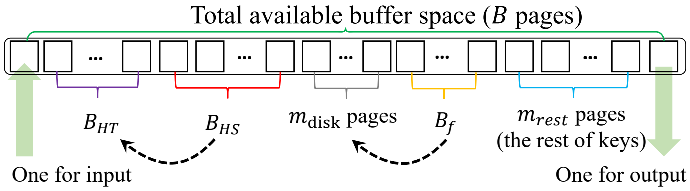


Fig. 6. The memory breakdown when partitioning R.

buffer is in total B pages, we need to ensure that Algorithm 8 uses no more than B pages. Figure 6 illustrates the memory breakdown.

- B_{HS}: #pages for the hash set HS_{mem} with **keys** in K_{mem};
- B_{HT}: #pages for the hash table HT_{mem} with **records** whose keys are in K_{mem};
- B_f: #pages for the hash map f_{disk} with **keys** in K_{disk} and their associated **partition identifiers**;
- m_{disk}: #pages as write buffer for all the on-disk partitions specified in f_{disk} (which is the same as #partitions specified in f_{disk});
- m_{rest}: #pages to partition records whose keys are in K_{rest};

Moreover, we denote ks as the size of each key (in bytes) and ps as the size of each page (in bytes). We can roughly have $B_{HT} = \lceil b_R \cdot |K_{\text{mem}}|/F \rceil$, $B_{HS} = \lceil ks \cdot |K_{\text{mem}}|/(F \cdot ps) \rceil$, and $B_f = \lceil (ks + 4) \cdot |K_{\text{disk}}|/(F \cdot ps) \rceil$, where a 4-byte integer is used to store the partition identifier in the hash map f_{disk}. At last, we come to the memory constraint subject to B (the total number of pages):

$$B_{HS} + B_{HT} + B_f + m_{\text{disk}} + m_{\text{rest}} \leq B - 2 \tag{3}$$

Cost function. Combining all these above together, we come to the cost function and integer programming as follows:

$$\begin{aligned}
\min_{K_{\text{mem}}, K_{\text{disk}}, \mathbb{P}_{\text{disk}}} \quad & \mathsf{Join}(K_{\text{disk}}, \mathbb{P}_{\text{disk}}, m_{\text{disk}}) + g_{\text{DHH}}(|K_{\text{disk}}|, m_{\text{rest}}) \\
\text{s.t.} \quad & B_{HS} + B_{HT} + B_f \leq B - 2 - m_{\text{disk}} - m_{\text{rest}}, \\
& 0 \leq m_{\text{disk}} \leq B - 2, 0 \leq m_{\text{rest}} \leq B - 2, \\
& K_{\text{mem}} \cap K_{\text{disk}} = \emptyset, K_{\text{mem}} \cup K_{\text{disk}} \subseteq K \\
& \sum_{j=1}^{m} \mathbb{P}_{i,j} = 1, \forall i \in [n] \\
& \mathbb{P}_{i,j} \in \{0, 1\}, \forall i \in [n], \forall j \in [m]
\end{aligned}$$

In the objective function, the first term, $\mathsf{Join}(K_{\text{disk}}, f_{\text{disk}}, m_{\text{disk}})$, is the join cost for K_{disk} using m_{disk} partitions, and the second term, $g_{\text{DHH}}(|K_{\text{rest}}|, m_{\text{rest}})$, is the estimated join cost of K_{rest} with m_{rest} pages using either GHJ or DHH, depending on how large m_{rest} is, as mentioned earlier. We will further discuss $g_{\text{DHH}}(|K_{\text{disk}}|, m_{\text{rest}})$ in Section 4.2. When estimating the DHH (GHJ) cost, we need to know the total number of records from S whose key falls into K_{rest}, which can be approximated by $n_S - \sum_{i \in K_{\text{mem}} \cup K_{\text{disk}}} \mathsf{CT}[i]$, since the CT values of keys in K are known from MCVs.

To find out the best combination of K_{mem}, K_{rest} and f_{disk}, we iterate over all possible values of $|K_{\text{mem}}|$, $|K_{\text{disk}}|$ and f_{disk}, and pick the one with the minimum estimated cost, as shown in Algorithm 10. Recall that once we have the ideal sizes of the sets, we use the CT values to populate them. Further, note that the optimal value of m_{rest} is calculated using Equation (3), since m_{disk} can be obtained from f_{disk}. Then, implied by the weakly-ordered property in Theorem 3.1 (i.e., keys with higher frequency should be kept in memory or in a small partition with higher priority), we always pick the top-$|K_{\text{mem}}|$ keys from K as K_{mem} and the next top-$|K_{\text{disk}}|$ keys from K as K_{disk}.

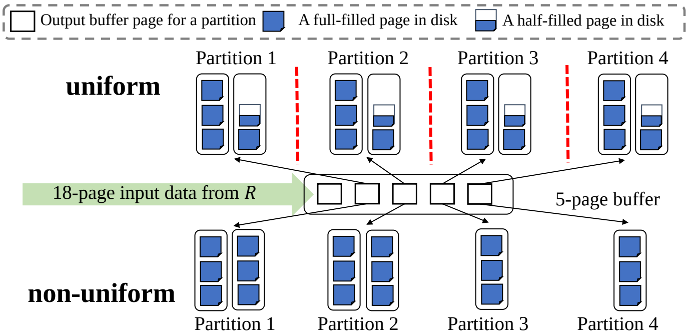


Fig. 7. An illustration of the uniform and non-uniform partitioning on the inner relation R, with a 5-page buffer (with 4 pages for partitioning R and 1 page for streaming the input). In NBJ, each chunk is up to 3-pages-long (with 1 page for streaming S and 1 page for the output assuming $F = 1.0$). Uniform partitioning leads to 4 partitions, each having $18/4 = 4.5$ pages worth of data. One needs to read S twice for each partition. As a comparison, rounded hashing with $n/c_R = 6$ and $m = 4$ generates 2 partitions with 6 pages, and 2 partitions with 3 page worth of data. In that way, partitions 1 and 2 need two passes on the corresponding partitions in S, while partitions 3 and 4 only need a single pass.

4.2 Optimizations with Rounded Hash

We now present another optimization for the partitioning phase, called *rounded hash* (RH), which may further reduce the cost of scanning the outer relation in the per-partition join. This is motivated by the divisible property in Theorem 3.1, according to which the sizes of all (but one) partitions should be divisible by c_R in the optimal partitioning. Figure 7 shows when this insight helps.

We note that uniform partitioning assigns records using a uniform hash function $\mathsf{PartID} = \mathsf{hash}(\mathsf{key}) \mod m$, based on the modulo function with respect to m (the number of partitions), which is denoted as *plain hash* (PH). To reduce the extra I/O cost using non-uniform partitioning, RH uses an additional modulo function with respect to $\lceil n/c_R \rceil$:

$$\mathsf{PartID} = (\mathsf{hash}(\mathsf{key}) \mod \lceil n/c_R \rceil) \mod m, \tag{4}$$

where $c_R = \lfloor b_R \cdot (B-2)/F \rfloor$ is the chunk size of records. Intuitively, RH maximizes the number of partitions with $\lfloor \lceil n/c_R \rceil / (B-1) \rfloor$ chunks. In addition to reducing unnecessary passes, RH also merges small partitions to avoid fragmentation [29].

Parametric Optimization. Since we rely on a hash function to partition records, partitions 1 and 2 in Figure 7 could occasionally *overflow* and grow larger than six pages due to randomness. In a skewed workload, overflown partitions will result in higher cost than an ideal uniform partitioning. A more robust scheme (Figure 7) is to assign 2.5 pages worth of records to partition 4 (so that an overflow would not spill to a new page) 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 (4), where β is in the range $(0, 1]$, and should be very close to 1 (we fix $\beta = 0.95$ in our implementation).

Cost Estimation. To get a rough estimation of how many I/Os are used by RH, we first build the I/O model for PH-based partitioning. We consider a generalized problem by estimating the per-partition cost for partitioning records positioned on the CT array from index s to index e into m partitions using PH, noted by $g_{PH}(s, e, m)$:

$$g_{\mathrm{PH}}(s,e,m) = \sum_{j=1}^{m} \mathbb{E}\left[\left\lceil \frac{|P_j|}{c_R} \right\rceil\right] \cdot \sum_{i \in P_j} \mathsf{CT}[i] = \left\lceil \frac{e-s+1}{m \cdot c_R} \right\rceil \cdot \sum_{i=s}^{e} \mathsf{CT}[i],$$

where each $|P_j|$ is approximated as a Poisson distribution with $\lambda = \frac{e-s+1}{m}$. When estimating the cost of RH, we need to know the proportion of data from S (noted by γ) that is scanned with one fewer pass ($1-\gamma$ of data needs one more pass):

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

As such, the normalized number of rounded passes is:

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

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

$$g_{\text{RH}}(s,e,m) = \#\text{rounded\_passes}(s,e) \cdot \sum_{i=s}^{e} \text{CT}[i] \tag{5}$$

The cost of our original problem is captured by $g_{\text{RH}}(1,n,m)$.

Overestimation using Chernoff Bound. When the fill ratio of each partition using PH reaches the predefined threshold β (that is, $\frac{e-s+1}{m} > \beta \cdot t \cdot c_R$, where t is the largest positive integer such that $t \cdot c_R > \frac{e-s+1}{m}$), we disable RH. We do this to avoid additional passes that may be triggered by occasional overflow in the hashing process. We now consider the impact of disabling RH on the cost model. Every element is distributed independently and identically using the same hash function, and thus, we apply the Chernoff bound to the overflow 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 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:

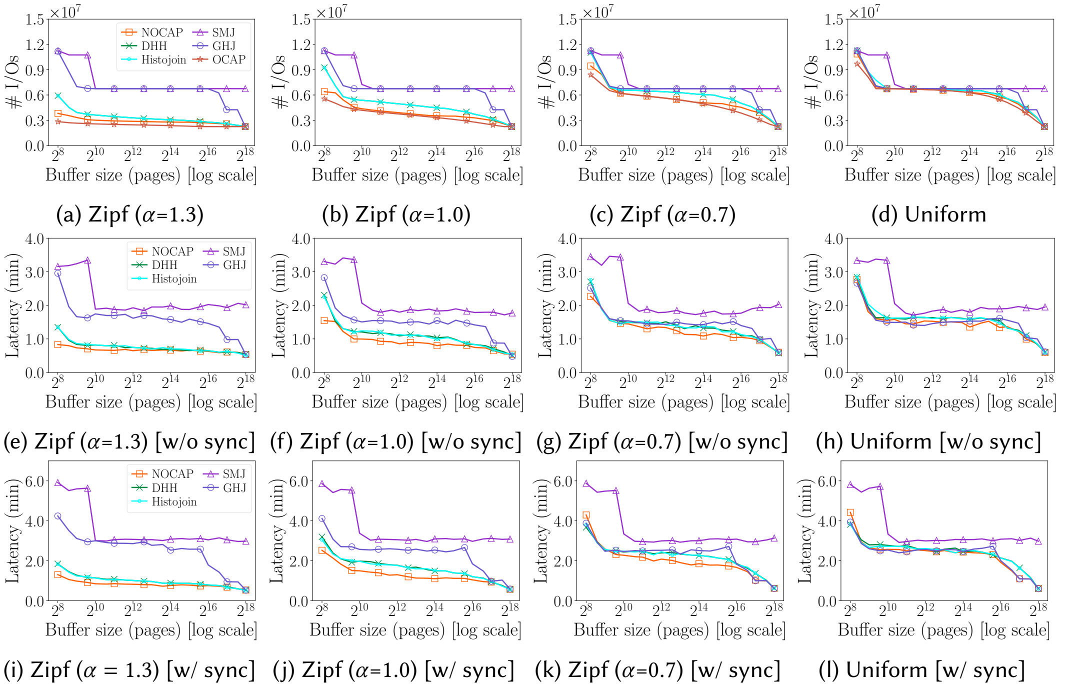
$$\#\text{rounded\_passes}(s,e) = \gamma \cdot \left\lceil \frac{e-s+1}{m \cdot c_R^*} \right\rceil + (1-\gamma) \cdot \left(\left\lceil \frac{e-s+1}{m \cdot c_R^*} \right\rceil + 1\right)$$

We update the $\#\text{rounded\_passes}(s,e)$ in Equation (5) to estimate the I/O cost.

5 EXPERIMENTAL ANALYSIS

We now present experimental results for our practical method.

Approaches Compared. We compare NOCAP with Grace Hash Join (GHJ), Sort-Merge Join (SMJ), and Dynamic Hybrid Hash join (DHH). For all the partitioning-based methods (GHJ, DHH, and NOCAP), we apply a lightweight optimization that picks the most efficient algorithm according to Table 1 for the partition-based joins. 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 (2% of the available memory is used for an in-memory hash table for skewed keys if the sum of their frequency is larger than 1% of the outer relation size). We compare NOCAP with Histojoin (by setting the trigger frequency threshold as zero). All the approaches are implemented in C++, compiled with gcc 10.1.0, in a CentOS Linux with kernel 4.18.0. For NOCAP, DHH, and Histojoin, we assume top $k = 50K$ frequently matching keys are tracked (i.e., 5% of the keys when $n = 1M$). In PostgreSQL, the most frequent keys are stored using a small amount of system cache, which does not consume the user-defined working memory budget unless they are inserted into the in-memory hash table when executing DHH. Similarly, in our prototype, the top-k keys are given. Due to the pruning techniques (§3.1.3), computing the partitioning scheme (Algorithm 10) with $k = 50K$ takes less than one second, so we omit its discussion.



(a) Zipf (α=1.3) (b) Zipf (α=1.0) (c) Zipf (α=0.7) (d) Uniform

(e) Zipf (α=1.3) [w/o sync] (f) Zipf (α=1.0) [w/o sync] (g) Zipf (α=0.7) [w/o sync] (h) Uniform [w/o sync]

(i) Zipf (α = 1.3) [w/ sync] (j) Zipf (α=1.0) [w/ sync] (k) Zipf (α=0.7) [w/ sync] (l) Uniform [w/ sync]

Fig. 8. While state-of-the-art skew optimization in DHH helps reduce the I/O cost, NOCAP better exploits join correlation knowledge to achieve even lower I/O cost and latency. The benefit of NOCAP is more pronounced with a skewed correlation.

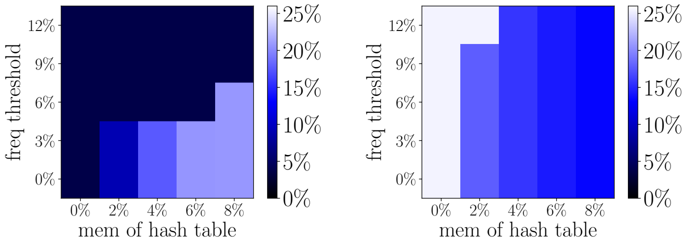
5.1 Sensitivity Analysis

We now present experimental results with synthetic data produced by our workload generator that allows us to customize the distribution of matching keys between two input relations to test the robustness of different join methods.

Experimental Setup. We use our in-house server, which is equipped with 375GB memory and two Intel Xeon Gold 6230 2.1GHz processors, each having 20 cores with virtualization enabled. As secondary storage, we use a 350GB PCIe P4510 SSD with direct I/O enabled. We can change the read/write asymmetry by turning the `O_SYNC` flag on and off. When `O_SYNC` is on, every write is first flushed to disk before returning and thus has higher asymmetry ($\mu_{sync} = 3.3$, $\tau_{sync} = 3.2$). When `O_SYNC` is off, we have lower asymmetry ($\mu_{no\_sync} = 1.28$, $\tau_{no\_sync} = 1.2$). By default, sync I/O is off to improve performance.

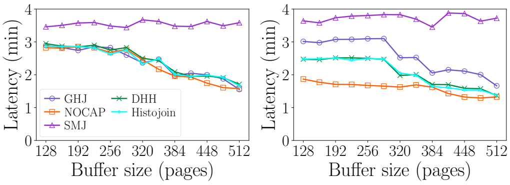
Experimental Methodology. We first experiment with a synthetic workload, which contains two tables R and S with $n_R = 1M$ and $n_S = 8M$. The record size is 1KB for both R and S, and thus we have $||R|| = 250K$ pages and $||S|| = 2M$ pages (the page size is fixed to 4KB). In this experiment, we vary 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 . Further, we run our experiments under uniform correlation and Zipfian correlation (short as Zipf) with $\alpha = 0.7, 1.0, 1.3$.

NOCAP Dominates for any Correlation Skew and any Memory Budget. For all the join methods, #I/Os decreases as the buffer size increases, as shown in Figures 8a-8d. We also observe that NOCAP achieves the lowest #I/Os (near-optimal) among all the join algorithms for any correlation



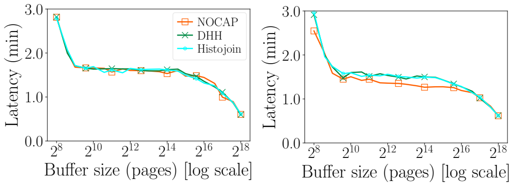
(a) $B = 2MB$ (b) $B = 32MB$

Fig. 9. DHH requires careful tuning to achieve its best performance, which is still slower than NOCAP.



(a) Uniform (b) Zipf (α=1.0)

Fig. 10. When the memory is limited, NOCAP can even reach 10% speedup with uniform workload.

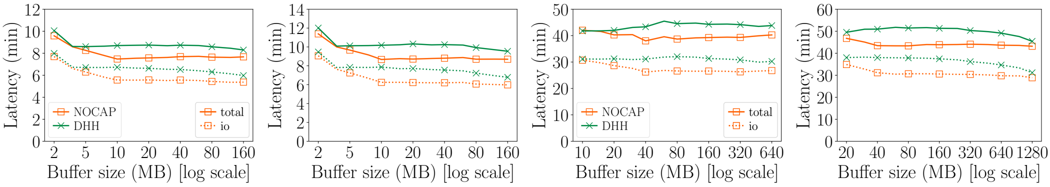


(a) Uniform (b) Zipf (α=0.7)

Fig. 11. With noisy MCVs, both DHH and NOCAP have similar performance to Figures 8h and 8g.

skew and any memory budget. When we compare latency from Figures 8e-8h, we conclude that GHJ and SMJ have a clear gap while they have nearly the same #I/Os regardless of the join correlation. This is because random reads in SMJ are 1.2× slower than sequential reads in GHJ. While these two traditional join methods have no optimization for skewed join correlations, DHH, Histojoin, and NOCAP take advantage of correlation skew to reduce #I/Os. However, DHH and Histojoin cannot fully exploit the join correlation because it limits the space for high-frequency keys with a fixed threshold (up to 2% of the total memory budget). Compared to DHH and Histojoin, 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 8f-8h show that the latency gap between NOCAP and DHH (Histojoin) is wider when we have medium skew ($\alpha = 1$ or $\alpha = 0.7$), compared to uniform. However, when the join correlation is highly skewed ($\alpha = 1.3$), the small hash table (2% of the available memory) in DHH is enough to capture the most frequent keys and, hence, yields near-optimal #I/Os (Figure 8a). As a result, the benefit of NOCAP for $\alpha = 1.3$ (Figure 8e) is smaller than for $\alpha = 1$ (Figure 8f). Nevertheless, NOCAP still issues 30% fewer #I/Os when the memory budget is small (2^8 pages) in the highly-skewed scenario.

DHH Cannot Adapt to Different Memory Budget. The skew optimization in DHH relies on two thresholds (memory budget and minimum frequency), and thus, it cannot easily adapt to a different memory budget. To verify this, we compare DHH with NOCAP under a given Zipfian correlation ($\alpha = 0.7$) using as memory budgets 2MB and 32MB. Figures 9a and 9b show the reduction of #I/Os when using NOCAP vs. DHH as a percentage of the DHH #I/Os, as we vary the two thresholds used by DHH. When B = 2MB, the best tuning for DHH (corresponding to the darkest cells) does not trigger skew optimization at all because the available memory is not big enough to have an impact. To do this, DHH either needs the memory budget for the skewed keys to be zero, or the frequency threshold to be very high. In contrast, when B = 32MB, the best DHH tuning assigns 8% memory to build the hash table for skewed keys, but it is still not able to outperform NOCAP. Note that even though DHH can achieve close-to-NOCAP performance by varying these two thresholds,



(a) σ_S=0.488, SF=10 (b) σ_S=0.63, SF=10 (c) σ_S=0.488, SF=50 (d) σ_S=0.63, SF=50

Fig. 12. TPC-H: NOCAP leads to higher speedup when more data from S are joined in a query.

this relies on workload-dependent, well-tuned parameters, which are hard to tune at runtime. In contrast, NOCAP is always able to find the best memory allocation.

NOCAP Outperforms DHH Even for Uniform Correlation with Small Memory Budget. We also examine the speedup when we have a lower memory budget. Although Figures 8l and 8h 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 10a, NOCAP achieves up to 15% and 10% speedup when there are 480 and 352 pages. The step-wise pattern of DHH 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 7. 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 10b.

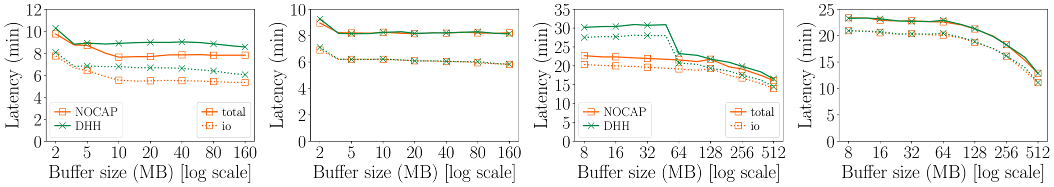
DHH and NOCAP Both Exhibit Robustness for Noisy MCVs. To examine how noisy CT can affect DHH and NOCAP, we add Gaussian noise to the CT values, with the average noise as 0 (the value of the average noise has no impact over MCVs) and $\sigma = n_S/n_R$. We reuse most of experimental settings so that $n_S/n_R = 8$ and thus $\mathrm{CT}_{noise}[i] \in [\mathrm{CT}[i] - 8, \mathrm{CT}[i] + 8]$ with probability 68%. We redo the experiments with uniform and Zipfian join correlation ($\alpha = 0.7$), and the results are shown in Figure 11. We do not observe a significant difference between the results with noisy CT values and the original results in Figures 8. The rationale behind this is that keys with high frequency are still likely to be prioritized (cached) during hybrid partitioning even after we add the Gaussian noise, thus having very similar performance to the results without noise.

5.2 Experiments with TPC-H, JCC-H, and JOB

We now experiment with TPC-H [53], JCC-H [10], and JOB [31], using DHH as our main competitor.

Experimental Setup. 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 skewed join 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 and cold keys, respectively, follows two different uniform distributions, which controls the overall skew and the average matching keys [12, 14]. In this experiment, we focus on a join correlation where 0.5% of the `o_orderkey` keys match 500 `lineitem` records on average, and the rest of the keys only match 1.5 `lineitem` records on average.

We further remove the filtering condition on `l_shipmode` and `l_receiptdate` so that the size of the filtered `lineitem` is larger than `orders` (since we focus on the case when $\|R\| < \|S\|$). We then have two conditions left, `l_receiptdate<l_shipdate` and `l_shipdate<l_commitdate`, with selectivity 0.488 and 0.63, respectively. We vary the selectivity by using either of the two conditions,



(a) JCC-H (Tuned Skew) (b) JCC-H (Original Skew) (c) JOB (cast ⋈ title) (d) JOB (cast ⋈ name)

Fig. 13. While DHH can perform as close as NOCAP, NOCAP is more adaptive when the workload varies.

thus experimenting with two selectivity levels (0.488 and 0.63), and with two scale factors (*SF*=10, *SF*=50), as shown in Figure 12. We employ storage-optimized AWS instances (*i3.4xlarge*) and turn off `O_SYNC`. The device asymmetry is $\mu = 1.2$ and $\tau = 1.14$.

NOCAP Accelerates TPC-H Joins. In Figure 12a, we observe that the difference in the total latency between NOCAP and DHH is not as large as what we saw in previous experiments. In fact, we find that the proportion of time spent on I/Os is lower in the TPC-H experiment due to the group by and aggregation in Q12. In earlier experiments (e.g., Figures 8f and 8g), the time spent on CPU is 10~20 seconds out of a few minutes (8% - 10% of the total latency). In contrast, we observe that in both Figures 12a and 12b, the proportion of time spent on CPU is more than 15%. This shows that the benefit of NOCAP reduces when we have more CPU-intensive operations in a query. In addition, Figure 12b shows that when we increase the selectivity on `lineitem`, the benefit of NOCAP also increases. We have a similar observation when we move to a larger data set (SF=50), as shown in Figures 12c and 12d. In fact, in both cases, when we have more `lineitem` data involved in the join, NOCAP has a higher speedup.

NOCAP Outperforms DHH for JCC-H and JOB. To further validate the effectiveness of NOCAP, we experiment with JCC-H (Join Cross Correlation) and JOB (Join Order Benchmark). JCC-H is built on top of TPC-H with adding join skew, so we reuse our TPC-H experimental setup (running the modified Q12 with $SF = 10$ and $\sigma_S = 0.488$). In the original JCC-H dataset, the majority (i.e., $300K \cdot SF$) of `lineitem` records only join with 5 `orders` records. To vary the skew, we tune the data generator so that there are around $5100 \cdot SF$ `orders` records, of which each matches 600 `lineitem` records on average. In addition to JCC-H, we also compare DHH and NOCAP by executing a PK-FK join between two tables from the JOB dataset. Specifically, `cast_info` stores the cast information between movies and actors, which are respectively stored in tables `title` and `name`. The number of movies per actor is highly skewed where the top 50 actors match 0.6% records in `cast_info`, while the number of actors per movie is less skewed where the top 50 movies match less than 0.1% records in `cast_info` ("top" in terms of the number of occurrences in `cast_info`). As we can observe in Figures 13a-d, when the correlation is extremely skewed (i.e., the original JCC-H and `cast_info` ⋈ `name`), DHH, that has fixed thresholds, exploits the skew in the join correlation to achieve close-to-NOCAP performance. However, when it comes to medium skew (i.e., tuned JCC-H and `cast_info` ⋈ `title`), NOCAP outperforms DHH.

6 DISCUSSION

General (Many-to-Many) Joins. In many-to-many joins, we have two CT lists that store the frequency per key in each relation, and thus, we need to create a new cost function (with two CT lists) to use as the main objective function. Despite that, we can still reuse our dynamic programming algorithm (Algorithm 5) and replace CalCost by considering extra CT values, however, the error bound is no longer guaranteed since our main theorem does not hold. In practice, even this crude approximation of the optimal partitioning for general joins is worth exploring since the CPU cost of enumerating partitioning schemes is very low, and it covers the design space of DHH with variable values for its two thresholds. As a result, NOCAP is still expected to outperform DHH for general joins. We leave the complete formulation for many-to-many joins as future work.

On-the-fly Sampling. On-the-fly sampling can give us more accurate CT values because it considers any selection predicate, while MCVs only rely on the estimated selectivity. However, relying on sampling without knowledge of MCVs makes the partitioning operation not pipelineable with the scan/select operator. This is because partitioning relies on the exact join correlation, but in the case of sampling, we need one complete pass first to acquire this information. In this case, we may downgrade NOCAP into DHH to avoid the additional pass. Another approach proposed by Flow-join [48] is to employ a streaming sketch to obtain an approximate distribution during partitioning. However, this requires scanning S first to build the histogram, which conflicts with partitioning R in the hybrid join. On the other hand, if we already have MCVs (note that it is acceptable to be noisy), on-the-fly sampling enables sideways information passing (SIP) [40]. Specifically, we can on-the-fly sample R during partitioning and build a Bloom filter (BF), which is later used when partitioning/scanning S. The additional BF makes the estimated CT more accurate, and thus, the partitioning generated by NOCAP is closer to the expected I/O cost. However, adding a BF introduces new trade-offs between performance and memory consumption.

7 RELATED WORK

In-Memory Joins. When the memory is enough to entirely fit both relations, in-memory join algorithms can be applied [5, 6, 34, 43, 50, 52]. 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 [22, 35, 49]. Although storage-based joins focus on #I/Os, when two corresponding partitions R_j and S_j both fit in memory, NOCAP can benefit from using the fastest in-memory join algorithm.

Distributed Equi-Joins. In a distributed environment, the efficiency of equi-joins relies on load balancing and communication costs. When the join correlation distribution is skewed, uniform partitioning may overwhelm some workers by assigning excessive work [54]. More specifically, when systems like Spark [57] rely on in-memory computations, the overall performance drops when the transmitted data for a machine does not fit in memory. Several approaches [7, 32, 48, 56] are proposed to tackle the data (and thus, workload) skew for distributed equi-joins.

8 CONCLUSION

In this paper, we propose a new cost model for storage-based partitioning PK-FK joins that allows us to find the optimal correlation-aware partitioning (OCAP) 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 suboptimal performance since it does not fully exploit the available memory and the join correlation information. To address the limitations of DHH, based on OCAP, we develop a practical near-optimal partitioning-based (NOCAP) join algorithm that supports variable memory constraints. We show that NOCAP dominates the state of the art for any available memory budget and for any join correlation, leading to up to 30% improvement.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. IIS-2144547, a Facebook Faculty Research Award, and a Meta Gift.

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 Ȯzsu. 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, Angelos-Christos G Anadiotis, and Steffen Kläbe. 2017. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *Performance Evaluation and Benchmarking for the Analytics Era - 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10661)*. 103–119. https://doi.org/10.1007/978-3-319-72401-0_8

[11] 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

[12] 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

[13] 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 Martín 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

[14] 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

[15] Cisco. 2018. Cisco Global Cloud Index: Forecast and Methodology, 2016–2021. *White Paper* (2018). https://tinyurl.com/CiscoGlobalCloud2018

[16] Bryce Cutt and Ramon Lawrence. 2009. Using intrinsic data skew to improve hash join performance. *Inf. Syst.* 34, 6 (2009), 493–510. https://doi.org/10.1016/j.is.2009.02.003

[17] Bryce Cutt, Ramon Lawrence, and Tolley Joshua. 2008. Discussion and Implementation of Histojoin in PostgreSQL. (2008). https://www.postgresql.org/message-id/flat/6EEA43D22289484890D119821101B1DF2C180E@exchange20.mercury.ad.ubc.ca

[18] 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

[19] 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

[20] Edsger W. Dijkstra. 1986. The strange case of the pigeon-hole principle. (1986). https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD980.PDF

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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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.

[31] 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

[32] 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

[33] 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

[34] 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

[35] Clemens Lutz, Sebastian Breßand 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

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

[37] MySQL. 2021. MySQL System Variables. (2021). https://dev.mysql.com/doc/refman/8.1/en/server-system-variables.html

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

[39] 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

[40] Laurel J Orr, Srikanth Kandula, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates Through Joins in Big-Data Clusters. *Proceedings of the VLDB Endowment* 13, 3 (2019), 252–265. https://doi.org/10.14778/3368289.3368292

[41] 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)*.

[42] 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).*

[43] 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

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

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

[46] PostgreSQL. 2023. Skew Optimization in PostgreSQL 16.0. (2023). https://github.com/postgres/postgres/blob/REL_16_0/src/include/executor/hashjoin.h#L121

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

[48] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. 2016. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE).* 1194–1205. https://doi.org/10.1109/ICDE.2016.7498324

[49] 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

[50] 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

[51] 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

[52] 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

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

[54] 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

[55] 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

[56] 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

[57] 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

[58] Zichen Zhu, Xiao Hu, and Manos Athanassoulis. 2023. NOCAP: Near-Optimal Correlation-Aware Partitioning Joins. arXiv:2310.03098 [cs.DB] https://aps.arxiv.org/abs/2310.03098

Received April 2023; revised July 2023; accepted August 2023