Towards Better Partitioning for Joins With Limited Memory

Zichen Zhu
zczhu@bu.edu
Boston University

Manos Athanassoulis
mathan@bu.edu
Boston University

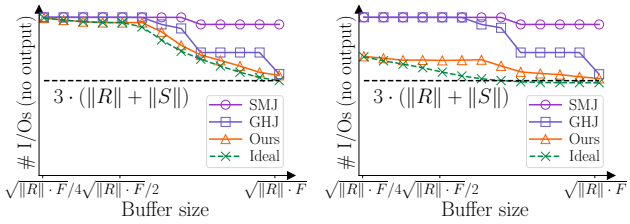
ABSTRACT

Hash joins are the de facto standard for implementing the join operator due to their efficient performance and judicious use of memory. This is particularly important when we operate under a limited memory budget requiring partitioning to and probing from storage, a common scenario in shared infrastructure like cloud-based data management. Typically, hash partitioning uniformly distributes all the records *regardless of the correlation of the join attributes* of the two input relations. Our analysis reveals that this results in unnecessary I/Os when a few join partitions are larger than the available memory. Since the state-of-the-art hash partitioning is uniform, if available memory is below $\sqrt{\textit{relation size}}$, most partitions will not fit in memory and would require a second round of hash partitioning, which leads to two main problems. First, this repartitioning can be partially avoided if we allow the initial partitions to have a variable size. Second, recursive hash partitioning writes data back to storage, which can be more expensive than reading in devices that have read/write asymmetry.

In this paper, we address these limitations of the state-of-the-art disk-based hash join by proposing three new ideas: (i) extensive use of **nested-block joins**, (ii) **Rounded Hash Join**, and (iii) **correlation-aware optimal join partitioning**. Nested-block joins avoid repartitioning when the size of the partition is just above the available memory and asymmetry would make the classical approach more expensive. Rounded Hash Join creates the maximum number of partitions that each fits in the available memory before creating larger partitions. Finally, the optimal partitioning algorithm finds the list-based partitioning that provably leads to the minimum number of I/Os when joining. We prove that the exact algorithm has polynomial complexity, and we approximate it to abide by the imposed memory constraints. Overall, our results show that our proposed suite of algorithms outperforms the state of the art by up to 1.8× under a limited memory budget, and is able to maintain near-optimal performance for skewed data while using 4× less memory than the state of the art.

1 INTRODUCTION

Relational equi-joins are ubiquitous in database systems. Recent research has focused on techniques to optimize in-memory joins [3–5, 7, 8, 20, 29, 32], however, resources might not always be sufficient to hold in memory two large tables simultaneously, thus requiring a classical disk-based join [28]. This is common in a shared resource setting, like multiple colocated databases, or virtual database instances that are deployed on the same physical cloud-resident server [6, 9, 18]. Further, in several data-intensive use-cases like Internet-of-things, edge computing, 5G communications, and autonomous vehicles [10, 15], memory might be even more limited. In any of the above use-cases, there are two more reasons that make storage-based joins important. First, when queries contain multiple joins, and second, as the number of concurrent queries increases, the existing resources must be shared among all concurrently executed join operators. This makes the available memory a resource shared among multiple instances of the join operator.



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

Figure 1: The number of I/Os for Grace Hash Join and Sort-Merge Joinn when they operate with limited memory increase to much more than the textbook ideal of $3 \cdot (\|R\| + \|S\|)$, however, in this paper, we prove that an optimal partitioning-based join needs much fewer I/Os, which our approach approximately matches. The benefit is significantly higher for a skewed correlation distribution. $\|R\|$ is the size of the smaller relation in pages, $\|S\|$ is the size of larger relation, and F is the hash table fudge factor.

Storage-based Join Algorithms. Traditionally, when memory is not enough to hold the hash table for either relation, Grace Hash Join (GHJ) is considered the state-of-the-art approach [12, 30]. In most cases, GHJ is preferred over Nested Block Join (NBJ) [16] and Sort-Merge Join (SMJ) because it is either outperforming them or they have similar performance, but GHJ makes judicious use of memory. Grace Hash Join uniformly assigns records from the two input relations to a number of partitions via hashing the join key, and the corresponding partitions are then joined. Specifically, when the smaller partition fits in memory, we simply store it in memory (typically as a hash table) and then scan the larger partition to produce the output. Note that many in-memory join algorithms can be used for the partition-to-partition join when they both fit in memory. However, if the smaller partition does not fit in memory, GHJ will recursively repartition both partitions, in an effort to create a new set of partitions that can be joined using the available memory. This repartitioning may be recursively repeated until for every pair of partitions (from the two tables), at least one fits in memory. A Nested-Block Join does not partition the data, rather, it partially loads the smaller relation (in **chunks** equal to the available memory) in the form of an in-memory hash table, and then scans the larger relation once *per chunk* to produce the join output for the

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

partial data. This process is repeated multiple times for the smaller relation until the entire relation is scanned. As such, the larger relation is scanned for as many times as the number of chunks in the smaller relation. Sort-Merge Join applies an external sort algorithm to sort the two input relations, which are then merged to produce the final join result. Since GHJ achieves its minimum cost when the buffer size is larger than $\sqrt{F \cdot \|R\|}$ (where $\|R\|$ is the number of pages of the smaller relation and F is the fudge factor), GHJ is usually preferred to other methods when we have constrained memory [16]. However, in this work, we identify that GHJ performs in some scenarios a high number of unnecessary I/Os leading to performance degradation. We outline the source of this behavior by introducing three problems that GHJ faces.

Problem 1: Recursive partitioning is expensive. When GHJ is joining two partitions, in case none of the partitions fit in memory, we will recursively partition both to create a set of smaller pairs of partitions. This process entails reading the partitions, writing them back to storage (to repartition), and eventually, reading them again while joining. Instead, the recursive repartitioning can be replaced by NBJ when the number of additional reads induced is smaller than the repartitioning cost. Further, consider that modern storage devices often exhibit a read/write asymmetry, where writes are more costly than reads [25, 26]. As a result, even when NBJ has a higher number of total I/Os, due to the asymmetry, it might be preferred over repartitioning. The exact decision depends on a simple cost model that takes into account asymmetry.

Problem 2: Uniform partitioning is sub-optimal. GHJ employs a random hash partition to distribute all the records, which are, thus, uniformly assigned to different partitions. Ideally, each partition fits in memory and thus only one pass is required to execute the join between two partitions. When memory is constrained and is below the ideal threshold of $\sqrt{F \cdot \|R\|}$, with a high probability, most partitions would not fit in memory as their sizes are approximately the same due to the uniformity of hash partitioning. However, if we allow partitions to have substantially different sizes, we may design a partitioning scheme that would have some partitions fit exactly in memory allowing for single-pass per-partition join, while other partitions will require more passes. This observation can help significantly reduce the total number of I/Os performed if we have fine granular control on how to partition the join relations.

Problem 3: GHJ does not exploit the correlation between two tables on the join key. In general, the join algorithms do not make any strong assumptions about the correlation between the join attributes. However, when we have a skewed correlation distribution, a few keys may join with the majority of the other relation. When multiple passes are required to perform the partition-wise join, joining the skewed keys may dominate the overall I/O performance. As a result, disregarding the join-key correlation when partitioning may lead to heavily sub-optimal partitioning resulting in a lot of excessive I/Os.

Our approach. We address the above three problems of GHJ by (i) extensive use of **nested-block** joins, (ii) the introduction of **Rounded Hash Join**, and (iii) a new **optimal correlation-aware join partitioning** strategy, respectively. First, we consider that the per-partition join kernel does not have to always be a recursive hash join. Instead, we consider all three GHJ, SMJ, and NBJ, and we observe that NBJ dominates the other join algorithms in most cases due to lower I/O cost and read/write asymmetry, and we use a simple cost-based decision to always find the optimal. Second, we propose a new hash partitioning scheme termed Rounded Hash Join, which allows variable-size partitioning to avoid sub-optimal uniform partitioning. The core idea is that instead of aiming for purely equal-size partitions, we aim for partitions that are equal in size, respecting available memory (so each partition fits in memory for future joining). If we need to add more entries once all partitions have reached this size, we only add entries to a subset of partitions, so that we maximize the number of partitions that fit entirely in memory. Third, we model the join cost as an optimization problem to exploit the correlation between two tables. This modeling assumes an arbitrary partitioning function (which we want to find) without any repartitioning. Essentially, the per-partition join mostly uses the NBJ kernel as discussed above. We propose a dynamic programming algorithm, termed *MatrixDP*, that finds the optimal solution in polynomial time, $O(n^2 \cdot m)$, a set of pruning techniques that further reduce this cost to $O\left(\left(n^2 \cdot \log m\right)/m^2\right)$ (where n is the number of records of the smaller relation and m is the memory budget in pages), as well as an approximate algorithm that further reduces this cost while strictly enforcing the memory constraints.

We implement Grace Hash Join, Sort-Merge Join, and our methods and integrate them into our prototype to directly compare them. Figure 1 shows the I/O count of GHJ, SMJ, and our method, along with the theoretic minimum under a memory budget which is less than $\sqrt{F \cdot \|R\|}$ for both uniform and Zipfian correlation between the two relations. A key observation is that both GHJ and SMJ perform more than the ideal number of I/Os, while our approach almost matches the ideal through the three techniques we propose. Further, we observe that the headroom for improvement is much higher for Zipfian correlation, for which equally sized partitions are suboptimal as we discuss in Section 3. Overall, our approach matches the ideal I/O cost, having the same number of I/Os with GHJ only with a fraction of the GHJ's memory requirement. Additionally, it only needs 25% of its memory requirement for Zipfian correlation. A second observation is that as we increase the available memory from $\sqrt{\|R\| \cdot F}/4$ to $\sqrt{\|R\| \cdot F}$, our approach gradually benefits, while GHJ only benefits in a step-wise manner, hence we can take advantage of more granular memory management.

Contributions. In summary, our contributions are as follows:

- We identify that NBJ can be extensively used in the probing phase to replace recursive GHJ. The benefit of NBJ is further magnified by read/write asymmetry (§2).
- We show that uniformly partitioning records under a limited memory budget results in excessive I/Os, and we propose *Rounded Hash Join* to alleviate the extra I/O cost (§3).
- We model the join cost assuming NBJ is applied in the probing phase and prove that obtaining an optimal partitioning scheme can be solved with polynomial complexity, even though the search space is exponential to the input size n (§4.1). We further propose two pruning methods that reduce the complexity to $O\left(\left(n^2 \cdot \log m\right)/m^2\right)$ (§4.2).
- We design an approximate partitioning algorithm based on the optimal algorithm to enforce memory budget constraints (§4.3).

Approach	estimated #I/O	Constraint
GHJ	$(2+\alpha) \cdot (\|\|R_i\|\| + \|\|S_i\|\|)$	$min(\|\|R_i\|\|, \|\|S_i\|\|) < (B-2)/F$
SMJ	$(2+\alpha) \cdot (\|\|R_i\|\| + \|\|S_i\|\|)$	$max(\|\|R_i\|\|, \|\|S_i\|\|) < (B-2)$
NBJ	$\|\|R_i\|\| + \frac{\|\|R_i\|\|}{(B-2)/F} \cdot \|\|S_i\|\|$	None

Table 1: Cost model for partition-wise join where $\|\|R_i\|\|$ and $\|\|S_i\|\|$ represent the number of pages from partition R_i and S_i.

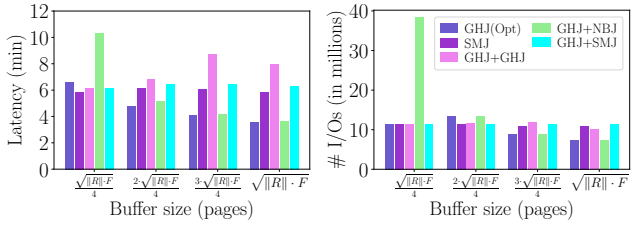
2 NESTED-BLOCK JOINS

We first re-consider the usefulness of NBJ. While NBJ can lead to a high I/O cost if we need a large number of passes (i.e., when the smaller relation has too many chunks), here we evaluate whether we can use NBJ as the join algorithm *after* the initial partitioning. In other words, we evaluate using NBJ as a building block of any partitioning-based algorithm. We now consider the state-of-the-art GHJ as the first layer of partitioning, however, we will refine this choice in Sections 3 and 4.

After GHJ creates the initial set of partitions, it may apply the same strategy (hash partitioning) recursively for pairs of join partitions that do not fit in memory. This scenario may happen when the memory is constrained. Consider a default MySQL [22] deployment where the smaller of the two input relations has 6300 pages (a 24MB relation) with a memory budget of 64 pages (the default memory budget is 256KB with pagesize 4KB). After the partitioning phase, each partition has ~100 pages on average, which is larger than the memory budget, hence, GHJ will repartition every partition, requiring reading and re-writing all data. Instead, NBJ might be a better candidate in this case since it will only need two passes of the outer relation, thus leading to lower total I/Os. In fact, either NBJ or SMJ can be used to replace recursive hash partitioning during partition-based joins. For ease of notation, GHJ+GHJ represents the standard recursive GHJ, and GHJ+NBJ and GHJ+SMJ employ NBJ and SMJ respectively after the partitioning phase in GHJ. SMJ stands for the standard SMJ (which is optimized by internal sort and merge join) and GHJ(Opt) uses the fastest strategy among GHJ, SMJ, and NBJ based on a simple cost model.

Table 1 summarizes the cost model. The α parameter denotes the read/write asymmetry [25], that is, a write is $\alpha\times$ slower than a read. $\|\|R_i\|\|$ and $\|\|S_i\|\|$ correspond to the size of each partition of R and S respectively after the first round of partitioning. For simplicity, we assume that each partition of relation R, R_i, is always smaller than the corresponding S_i. When memory is not enough to hold each R_i partition, GHJ(Opt) chooses the strategy with the minimum cost using the cost model from Table 1 to execute the partition-wise join. As the partition sizes may differ, the optimal strategy can also differ for different partitions. Note that, for GHJ, after the initial partitioning phase, we only consider one more round of recursive re-partitioning, since the partitions would already have a very small size. In addition, when the I/O cost is the same between SMJ and GHJ, SMJ is preferred because the writes of SMJ are sequential and hence faster than the random writes of GHJ.

Experimental Setup. We now run a microbenchmark to showcase that NBJ can be used as a building block of partitioned-based joins. We run our experiments in our in-house server, which is equipped with two Intel Xeon Gold 6230 2.1GHz processors and each has 20 cores with virtualization enabled. For our storage, we use a PCIe P4510 SSD with direct I/O enabled in our experimentation.



(a) Latency **(b) I/O count**

Figure 2: GHJ+NBJ yields lower latency when the buffer size is larger than or equal to $\sqrt{\|\|R\|\| \cdot F}/2$, while GHJ+SMJ performs better when the buffer size becomes more constrained

Workloads and Metrics. We join table R that has 1 million records with unique keys ($n_R = 1M$) with table S that has 8 million records ($n_S = 8M$). We assume that the join key is also the primary key of R, thus $n = n_R$ and the average number of matching records for each unique key is 8. . In addition, the key size is 8 bytes and the entry size for both relations is 1KB. The size of the two relations are $\|\|R\|\| = 250K$ and $\|\|S\|\| = 2M$, thus, the memory budget that can achieve the minimum cost for GHJ is $\sqrt{F \cdot \|\|R\|\|} \approx 505$. We vary the memory budget from 128 to 512 pages (0.5 MB to 2 MB). To report the exact join latency, we discard the join output buffer page once it is full, not including the cost of writing the join result.

Results and Analysis. While traditional GHJ uses recursive GHJ when a partition does not fit in memory, our results show that recursive GHJ is beneficial only when the memory buffer is extremely small ($< \sqrt{\|\|R\|\| \cdot F}/4$ pages). As shown in Figure 2, GHJ(Opt) chooses NBJ when the buffer size is $\geq \sqrt{\|\|R\|\| \cdot F}/4$ and thus achieves similar performance to GHJ+NBJ. Specifically, when the buffer size has $\sqrt{\|\|R\|\| \cdot F}/2$ pages, we should have chosen SMJ instead, considering the I/O count, as shown in Figure 2b (the number of I/Os from NBJ is higher than GHJ+SMJ and GHJ+GHJ). However, taking into account the read/write asymmetry, even though NBJ involves more reads, each read is $\alpha\times$ cheaper than the write operation (4.5× in our device) and thus NBJ is chosen as the best strategy in GHJ(Opt) outperforming both GHJ+SMJ and GHJ+GHJ.

This microbenchmark motivates our decision for extending GHJ and other partitioning-based join algorithms with a lightweight join optimizer that considers read/write asymmetry before deciding whether to repartition, use SMJ, or use NBJ. In fact, in the majority of the cases, NBJ is the algorithm of choice, hence our techniques in the rest of the paper focus on optimizing the NBJ cost in the probing phase of the partition-based join.

3 ROUNDED HASH JOIN

Traditional hash partitioning distributes records to each partition uniformly, but this may incur excessive I/Os when the memory is not sufficient. Consider the example in Figure 3. When we have 5 pages in the buffer, we can produce at most four partitions since one page is used for streaming the input. In the top part of the figure, uniform hash partitioning creates 4 partitions each having $18/4 = 4.5$ pages worth of data (for illustration, we fix the fudge factor as 1.0). As we discussed in the previous section, NBJ is preferable to repartitioning because it avoids the expensive re-writing of the data. However, if we apply NBJ, the maximum chunk we load at a

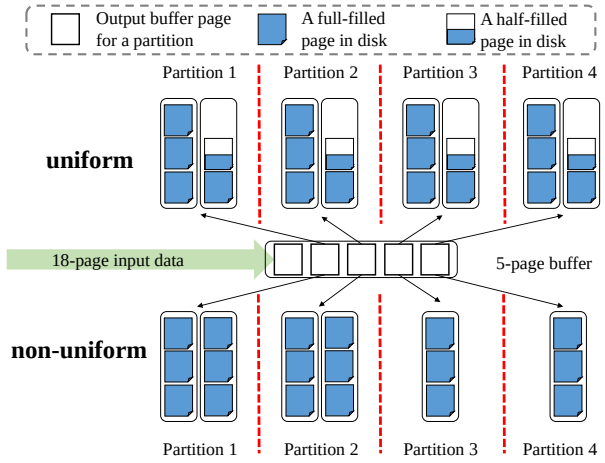


Figure 3: For uniform partitioning, none of the partitions fits in memory and thus all the records from the second relation are scanned twice. However, for non-uniform partitioning, only partitions 1 and 2 from the second relation are scanned twice.

time is 3-page long since we need one page for streaming the outer relation and one page for the output. As a result, in the example shown, for each of the four partitions we will read the outer relation twice (i.e., 2 chunks per partition). However, we can avoid some of these accesses if we allow for a non-uniform partitioning as shown at the bottom of the figure. We next discuss how to achieve this.

Formally, uniform partitioning assigns records using

$$PartID = hash(key) \mod m \tag{1}$$

where m is the number of partitions. When memory is constrained, we have $m = B - 1$, where B is the memory budget in pages. To reduce the extra I/O cost using non-uniform partitioning, we propose a **Rounded** Hash Join as shown in Equation (2).

$$PartID = \left(hash(key) \mod \left\lceil \frac{n}{\lfloor b_R \cdot (B-2)/F \rfloor} \right\rceil \right) \mod m \tag{2}$$

where b_R represents the number of records per page for relation R when building the hash table. Since b_R, B, F are all constant numbers, we define the chunk size as a constant $c_R = \lfloor b_R \cdot (B-2)/F \rfloor$. In the example shown in Figure 3, the number of total chunks n/c_R is 6 and the number of partitions m is 4. Applying the *mod 6* operation, keys with a hash value remainder 0 or 4 are assigned to the first partition, keys with a hash value remainder 1 or 5 are assigned to the second partition and the remaining keys are evenly distributed in partitions 3 and 4. In that way, when joining partitions 1 and 2 with NBJ, we will perform two passes on the corresponding inner partitions, while partitions 3 and 4 will only lead to a single pass. In fact, we maximize the number of partitions that have $\lfloor \lceil n/c_R \rceil / (B-1) \rfloor$ chunks. For example, if we uniformly partition 30 pages using a 5 page buffers, each partition will be 7.5 pages long (which corresponds to 3 chunks). Using RHJ, we will get 2 partitions with 9 pages (3 chunks each) and 2 partitions with 6 pages (2 chunks each, $\lfloor (30/3)/4 \rfloor = 2$). Note that for all partitions with 2 chunks, during the per-partition join, we will scan the corresponding partition from the inner relation only twice.

4 CORRELATION-AWARE JOIN

Problem Statement and Basic Notation. We now consider the problem of finding the optimal join partitioning when we have knowledge of the correlation between the join attributes. We consider *primary key / foreign key* (PKFK) joins for which, we quantify the correlation between the join attributes using a *correlation table* CT, s.t., $CT[i]$ represents the number of matching records in the outer relation S for the i^{th} record in the inner relation R. Note that in our notation, we assume that R's join attribute is the primary key, and S's join attribute is the foreign key. Further, for simplicity, we assume w.l.o.g. that R is the smaller relation that is used to build the in-memory hash table (the derived results hold even when S is the smaller relation). Our goal is to find the partitioning of join keys that would minimize the total I/Os for the join execution.

To achieve this, we first formulate an objective function and an *optimization problem*. We then design a dynamic programming solution (§4.1), augment it with two more pruning techniques (§4.2), and, finally, propose an approximate partitioning algorithm to limit memory consumption (§4.3).

4.1 Minimizing Data Accesses

Problem Definition. Any partitioning scheme is essentially an assignment of n records into m partitions. Since we consider PKFK joins, once we know how to partition R (the table with the primary key), we can apply the same partitioning scheme to S. We represent a partitioning scheme with a binary $n \times m$ partition matrix P, where $P_{i,j} = 1$ means that the i^{th} record belongs to the j^{th} partition. Every record must belong to exactly one partition, hence:

$$\forall i \in [n], \quad \sum_{j=1}^{m} P_{i,j} = 1 \tag{3}$$

Similar to GHJ, in the partitioning phase, we use one page to stream the input relation and the rest of the pages as output buffers for each partition, hence, $m \leq B - 1$. Then we join the partitions pair-wise. Using the notation of the partition matrix, the size (in pages) of a partition R_j is given by $\|R_j\| = \sum_{i=1}^{n} P_{i,j}/b_R$, and the size of the corresponding partition S_j is given by $\|S_j\| = \sum_{i=1}^{n} P_{i,j} \cdot CT[i]/b_S$, where b_R (b_S) is the number of records per page for R (S). We now consider that the partition-wise join will use NBJ, and in order to quantify the cost of the join, we use the number of passes to calculate the *per-partition join cost* for the j^{th} partitions R_j with S_j:

$$\|R_j\| + \#\text{passes} \cdot \|S_j\| = \|R_j\| + \left\lceil \frac{\|R_j\|}{(B-2)/F} \right\rceil \cdot \|S_j\| \tag{4}$$

The total *join cost* can now be given as follows.

$$JC = \sum_{j=1}^{m} \left(\|R_j\| + \left\lceil \frac{\|R_j\|}{(B-2)/F} \right\rceil \cdot \|S_j\| \right) \tag{5}$$

Since the first part of the equation always calculates $\|R\|$, we simplify the cost function to only include the number of pages accessed from the S partition. Further, we assume that the initial partitioning cost is constant since it has to read and write each relation. We now formulate the optimization problem where the objective function is $JC(P, CT)$, where the correlation table CT is provided and the

output is the partitioning (in the form of a partitioning matrix P).

$$\begin{aligned} &arg\min_{P} JC(P, CT) \\ &s.t.\ \forall i \in [n], \quad \sum_{j=1}^{m} P_{i,j} = 1 \end{aligned} \tag{6}$$

$$where\ JC(P, CT) = \sum_{j=1}^{B-1} \left(\left\lceil \frac{\|R_j\|}{(B-2)/F} \right\rceil \cdot \|S_j\| \right) \Rightarrow$$

$$JC(P, CT) = \sum_{j=1}^{B-1} \left(\left\lceil \frac{\sum_{i=1}^{n} P_{i,j}}{c_R} \right\rceil \cdot \left(\sum_{i=1}^{n} P_{i,j} \cdot CT[i] \right) \right) \tag{7}$$

We then rearrange the optimization function as follows:

$$JC(P, CT) = \sum_{i=1}^{n} CT[i] \cdot \left(\sum_{j=1}^{B-1} P_{i,j} \cdot \left\lceil \frac{\sum_{l=1}^{n} P_{l,j}}{c_R} \right\rceil \right) \tag{8}$$

Preliminaries. For ease of notation, we note $[n]$ as the set $\{i | i \in \mathbf{N}, i \leq n\}$ and $[a, b]$ as the set $\{i | i \in \mathbf{N}, a \leq i \leq b\}$. Also, we assume CT is already sorted in ascending order of $CT[i]$. In the implementation, we may just store together $CT[i]$ with the real index i^* (i.e., the row number of the actual table R). Besides, with the constraint that $\sum_{j=1}^{B-1} P_{i,j} = 1$, we know that for each record, there exists exactly one partition so that $P_{i,j} = 1$. Suppose that we use a function f to define the mapping relation between the record index i and the index of partition $f(i)$ where it belongs. We can then rewrite the join cost as follows:

$$JC(P, CT) = \sum_{i=1}^{n} CT[i] \cdot \left\lceil \frac{\sum_{l=1}^{n} P_{l,f(i)}}{c_R} \right\rceil = \sum_{i=1}^{n} CT[i] \cdot \left\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \right\rceil \tag{9}$$

Observe that $\sum_{l=1}^{n} P_{l,f(i)}$ corresponds to the number of records that are in the same partition with record i. We denote this set of records as $\mathcal{N}(i)$. By definition, we have $i \in \mathcal{N}(i)$ and thus $\|\mathcal{N}(i)\| \geq 1$, $\left\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \right\rceil \geq 1$. Alternatively, we also use $\mathcal{P}_j (1 \leq j \leq m)$ to represent the record set for j^{th} partition. By definition, we have

$$P_{i,j} = 1 \Leftrightarrow f(i) = j \Leftrightarrow \mathcal{N}(i) = \mathcal{P}_j$$

Obtaining the Optimal Partitioning. Observe from Equation (9) that the overall join cost can be also calculated by aggregating $CT[i] \cdot \left\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \right\rceil$ for each record. Given that CT is already sorted, we can take advantage of this so that the join cost can be reduced. This intuition leads to Lemma 4.1.

Lemma 4.1. *For a given partitioning P where there exist two record indexes i_1, i_2 which satisfy $i_1 > i_2$ and $\left\lceil \frac{\|\mathcal{N}(i_1)\|}{c_R} \right\rceil > \left\lceil \frac{\|\mathcal{N}(i_2)\|}{c_R} \right\rceil$, we can always obtain a new partition P' with a lower or equivalent join cost by swapping i_1 and i_2 (i.e., $JC(P', CT) \leq JC(P, CT)$).*

Proof. First, i_1 and i_2 must belong to different partitions (i.e., $f(i_1) \neq f(i_2)$), otherwise $\left\lceil \frac{\|\mathcal{N}(i_1)\|}{c_R} \right\rceil = \left\lceil \frac{\|\mathcal{N}(i_2)\|}{c_R} \right\rceil$ which violates the prerequisite. Observe from Equation (9), for all the records except i_1, i_2, the join cost does not change (we use the primary key constraint to ensure that there are no duplicate keys for i_1 and i_2 and thus only two records are swapped), even after we swap the positions of i_1 and i_2 (we note the new partition matrix as P'). This is because $\forall i \in \mathcal{N}(i_1)/\{i_1\}$, $\|\mathcal{N}(i_1)\|$ does not change since i_2 takes place of i_1 and the same for all the records in $\mathcal{N}(i_2)/\{i_2\}$. As such, we have:

$$\begin{aligned} &JC(P', CT) - JC(P, CT) = \\ &CT[i_1] \cdot \left\lceil \frac{\|\mathcal{N}(i_2)\|}{c_R} \right\rceil + CT[i_2] \cdot \left\lceil \frac{\|\mathcal{N}(i_1)\|}{c_R} \right\rceil \\ &- CT[i_1] \cdot \left\lceil \frac{\|\mathcal{N}(i_1)\|}{c_R} \right\rceil - CT[i_2] \cdot \left\lceil \frac{\|\mathcal{N}(i_2)\|}{c_R} \right\rceil \end{aligned} \tag{10}$$

Note that CT follows an ascending order, and thus $CT[i_1] \geq CT[i_2]$. At the same time, we know $\left\lceil \frac{\|\mathcal{N}(i_1)\|}{c_R} \right\rceil > \left\lceil \frac{\|\mathcal{N}(i_2)\|}{c_R} \right\rceil$.

$$\begin{aligned} &CT[i_1] \cdot \left\lceil \frac{\|\mathcal{N}(i_2)\|}{c_R} \right\rceil + CT[i_2] \cdot \left\lceil \frac{\|\mathcal{N}(i_1)\|}{c_R} \right\rceil \\ &- CT[i_1] \cdot \left\lceil \frac{\|\mathcal{N}(i_1)\|}{c_R} \right\rceil - CT[i_2] \cdot \left\lceil \frac{\|\mathcal{N}(i_2)\|}{c_R} \right\rceil \\ = \quad &(CT[i_1] - CT[i_2]) \cdot \left(\left\lceil \frac{\|\mathcal{N}(i_2)\|}{c_R} \right\rceil - \left\lceil \frac{\|\mathcal{N}(i_1)\|}{c_R} \right\rceil \right) \leq 0 \\ \Leftrightarrow \quad &JC(P', CT) \leq JC(P, CT) \end{aligned} \tag{11}$$

□

Note that his lemma implies that there is an optimal solution P_{opt} (even if it might be expensive to find it). Using Lemma 4.1, we can have the following theorem:

Theorem 4.2. *There exists an optimal partitioning scheme P_{opt} where each partition contains consecutive elements from CT.*

Proof. We use **strong induction** on the number of partitions to prove Theorem 4.2. We note $Prop_m$ as the proposition for Theorem 4.2 when the number of partitions is fixed as m. First, $Prop_1$ is true because when the number of partitions is 1, no matter how many input elements we have, they belong to a single partition, and thus they are naturally consecutive elements from CT. Assume that $Prop_1, Prop_2, ..., Prop_{m-1}$ are all true for every possible n and CT. Next we need to prove $Prop_m$. For every possible input dataset, we can obtain an optimal partitioning scheme P by enumeration theoretically and choose one partition $\mathcal{S}$ from P. We apply $Prop_{m-1}$ to re-partition all the elements apart from $\mathcal{S}$, and we then get $m-1$ partitions of which all elements are consecutive. Next, we show that there exists a process that can obtain a new partitioning scheme P' where all the partitions in P' only contain consecutive elements without increasing the join cost.

As shown in Figure 4, $\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_{m-1}$ (consecutive gray rectangles separated by a red line representing one partition) represent $m-1$ partitions that contain consecutive records and the remaining records are grouped together and form the m^{th} partition $\mathcal{S} = [n] - \bigcup_{j=1}^{m-1} \mathcal{P}_j$. By $Prop_{m-1}$, we ensure that all the $m-1$ partitions (i.e., $\{\mathcal{P}_j | j \in [m-1]\}$) contain consecutive records. Note that containing consecutive records in $[n] - \mathcal{S}$ does not necessarily mean they have to be consecutive in $[n]$ (e.g., $\mathcal{P}_1$ and $\mathcal{P}_2$ in Figure 4 can also belong to one consecutive partition from $Prop_{m-1}$), but

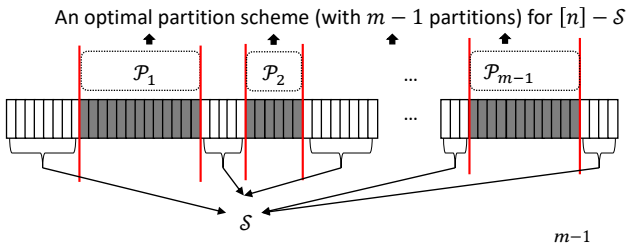


Figure 4: $m-1$ consecutive partitions and $\mathcal{S} = [n] - \bigcup_{j=1}^{m-1} \mathcal{P}_j$

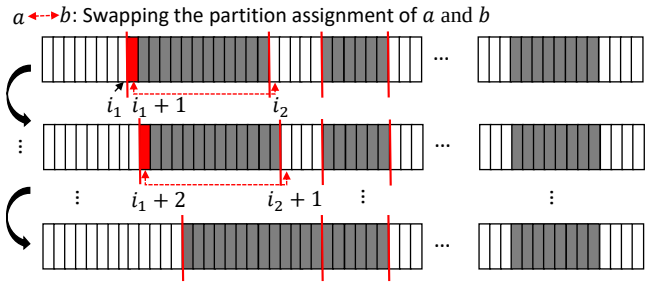


Figure 5: When $\|\mathcal{P}_1\| \le \|\mathcal{S}\|$ ($\mathcal{P}_1 = [i_1+1, i_2-1]$), and there is only one partition $\mathcal{P}_1$ for the range $[i_1+1, i_2-1]$, we swap partitioning assignment as above to make elements consecutive in the m^{th} partition without increasing $JC(P, CT)$.

the following process still applies in that case. As shown in Figure 5, when $\mathcal{S}$ contain non-consecutive elements, we can always find two indexes i_1, i_2 ($i_1 + 1 < i_2$ and $i_2 \in \mathcal{S}$) so that $[i_1+1, i_2-1]$ do not belong to $\mathcal{S}$ (i.e., $\forall i \in [i_1+1, i_2-1]\ i \notin \mathcal{S}$)). If $\|\mathcal{P}_1\| \le \|\mathcal{S}\|$, according to Lemma 4.1, we can swap the partition assignment of i_1+1 and i_2 to obtain a new partition without increasing the cost. Note that there could be more than one consecutive partitions for elements from $[i_1+1, i_2-1]$. In this case, some partitions may contain nonconsecutive elements after swapping, as shown in Figure 6. We can then apply $Prop_k$ to re-partition $\bigcup_{j=1}^{k} \mathcal{P}_{j_k}$. Following the above process, we can continue swapping elements until no more elements from $\mathcal{S}$ have larger indexes than any element in $\mathcal{P}_1$. In fact, we can apply this process to all the partitions of which the size is less than $\|\mathcal{S}\|$. At the end, all smaller partitions will be placed after $\mathcal{S}$. Similarly, we can also do this for all larger partitions. As shown in Figure 7, we swap the partition assignments between $i_1 - 1$ and i_2 to move $\mathcal{P}_1$ forward until $\mathcal{P}_1$ is placed before $\mathcal{S}$. When there are multiple consecutive partitions for the range $[i_1+1, i_2-1]$, we can apply $Prop_k$ to re-partition data without increasing the cost. After the above process, all larger partitions are placed before $\mathcal{S}$ and all smaller or equal-size partitions are placed after $\mathcal{S}$. All elements in $\mathcal{S}$ are consecutive now. Proof completes. □

Figure 4 through Figure 7 provide a visual description of the proof. According to **Theorem 4.2**, we can now reduce the search complexity using dynamical programming. **Instead of iterating all the possible combinations, we only need to look for the optimal solution when all the partitions contain consecutive elements in the CT.** To find the optimal partitioning scheme for (i, j) (i is the number of elements to be partitioned and j is the number of partitions) and the CT, we iterate all possible cutting

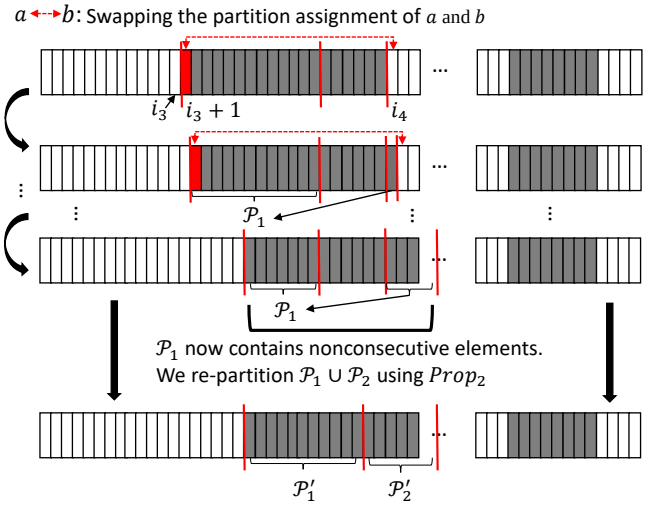


Figure 6: When $\|\mathcal{P}_1\| \le \|\mathcal{S}\|$ and there are two partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ for the range $[i_3+1, i_4-1]$, we swap partition assignments between i_3+1 and i_4, which results in nonconsecutive $\mathcal{P}_1$. We re-partition $\mathcal{P}_1 \cup \mathcal{P}_2$ using $Prop_2$ to obtain two new consecutive partitions $\mathcal{P}_1'$ and $\mathcal{P}_2'$ without increasing the overall join cost $JC(P, CT)$.

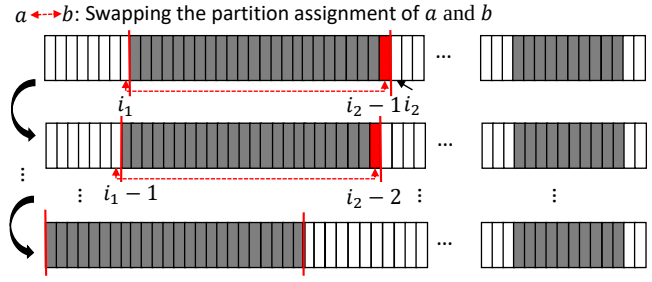


Figure 7: When $\|\mathcal{P}_1\| > \|\mathcal{S}\|$ and there are two partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ for the range $[i_1+1, i_2-1]$, we swap partition assignments between i_1 and i_2-1 using Lemma 4.1. We can continue to swap two elements until there are no elements from $\mathcal{S}$ that can be replaced, that is, $\mathcal{S}$ is placed after $\mathcal{P}_1$.

positions between 1 and i and select the cutting position with the minimum cost. We can store the results into a matrix to avoid repeated computation. We propose a dynamic programming algorithm, called MatrixDP, shown in detail in Algorithm 1. The object Cut in $V[i][j]$ stores the minimum cost ($cost$) and the cutting position ($lastPos$) (the start index of the last partition) for the first i elements with j partitions.

Note that, $CalCost(start\_idx, end\_idx)$ (short as $CalCost(s, e)$) can be easily computed using the pre-computed prefix sum (PPS):

$$PPS[k] = \sum_{i=1}^{k} CT[i],$$

and this only requires scanning CT once. Therefore, when we calculate the cost, we can reuse PPS as follows:

$$CalCost(s, e) = (PPS[e+1] - PPS[s+1]) \cdot \left\lceil \frac{e-s+1}{c_R} \right\rceil \quad (12)$$

Algorithm 1 MatrixDP(CT, n, m)

```
1: V ← Cuts[n+1][m+1]
2:       ▷ Cut stores the minimum cost and the last cut for (i, j)
3: for i ← 0 to n do
4:     V[i][1].cost ← CalCost(0, i)
5:     V[i][1].lastPos ← 0
6: end for
7: for i ← 1 to n do
8:     for j ← 2 to m do
9:         V[i][j].cost ← MAX_INT
10:        for k ← 0 to i − 1 do
11:            tmpCost ← V[k][j − 1].cost + CalCost(k + 1, i)
12:            if tmpCost < V[i][j].cost then
13:                V[i][j].cost ← tmpCost
14:                V[i][j].lastPos ← k + 1
15:            end if
16:        end for
17:    end for
18: end for
19: return V
```

We can backtrack the index of the last partition to complete all the cutting positions, as shown in Algorithm 2.

Algorithm 2 *Get_Cutting_Pos*(V, n, m)

```
CutPositions ← {}
lastPos ← n
for i ← 0 to m − 2 do
    lastPos ← V[lastPos][m − i].lastPos − 1
    CutPositions.push_back(lastPos + 1)
end for
return CutPositions.reverse()
```

Complexity Analysis. Since $CalCost()$ has a constant cost, the overall computation cost comes from three for-loops. Observe that we are iterating the range $[0, i-1]$ and $i \in [0, n]$, the total complexity is $O(m \cdot n^2) \ll O(m^n)$. Since we have a large matrix to store the intermediate results, the space complexity is $O(m \cdot n)$.

4.2 Pruning

We further propose two pruning techniques: *monotonicity-based pruning* and *step-based pruning*, to lower the time and space complexity. The core idea of these two pruning techniques is to avoid the unnecessary computation of $V[i][j] (i \in [n-1], j \in [m-1])$ in Algorithm 1 when they are not useful for computing $V[n][m]$.

4.2.1 Monotonicity-based Pruning. The proof of **Theorem 4.2** shows that, for a given partitioning scheme $\mathcal{P}$, we can move all larger partitions before $\mathcal{P}$ and all smaller partitions after $\mathcal{P}$. In fact, we can have **Corollary 4.2.1** based on **Theorem 4.2**.

Corollary 4.2.1. *There exists an optimal partitioning scheme P where all the partitions contain consecutive elements w.r.t. their CT value, and the partition size sequence $\{\|\mathcal{N}(i)\|\}$ is in decreasing (monotonic non-increasing) order.*

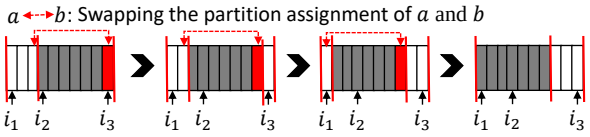


Figure 8: When there are two adjacent partitions which do not follow the decreasing order, we can always apply Lemma 4.1 to swap elements. For example, in the first step, we have $i_3 > i_2 - 1$ and $\|\mathcal{N}(i_3)\| > \|\mathcal{N}(i_2 - 1)\|$, and thus $\left\lceil \frac{\|\mathcal{N}(i_3)\|}{c_R} \right\rceil \geq \left\lceil \frac{\|\mathcal{N}(i_2-1)\|}{c_R} \right\rceil$, we can swap i_3 and i_2-1. We continue swapping elements until the indexes of all the elements in the larger partition are smaller than the other partition.

Proof. We use the optimal partitioning scheme derived from **Theorem 4.2**. Suppose that we find two adjacent partitions that are not in the non-decreasing order, as shown in Figure 8. We can apply **Lemma 4.1** multiple times until we have the larger partition is completely placed before the smaller one. We can repeat this process for every two adjacent partitions until all the partitions are decreasingly sorted by their size. Note that, we always ensure that the join cost does not increase by Lemma 4.1 and thus the final partitioning scheme is still optimal. Proof completes. □

Note that, according to **Corollary 4.2.1**, records with $CT[i] = 0$ should be assigned to the largest partition to minimize the join cost. However, we actually do not need to partition these records since they do not have matching records. Therefore, we can safely skip records with $CT[i] = 0$ by pre-processing CT to reduce the input size of Algorithm 2.

Combing Corollary 4.2.1 with pigeonhole principle [13], we can derive **Corollary 4.2.2**.

Corollary 4.2.2. *The optimal partitioning scheme P from **Corollary 4.2.1** should satisfy: 1) the largest partition contains at least the first $\lfloor \frac{n-1}{m} \rfloor + 1$ elements in CT, and 2) the smallest partition contains at most the last $\lfloor \frac{n}{m} \rfloor$ elements in CT.*

Note that **Corollary 4.2.2** holds for every (n, m) pair, which implies that, for every (i, j) pair in the MatrixDP(CT, n, m) algorithm, we can apply this pruning at line 10 ($k \leftarrow 0$ to $i-1$) in Algorithm 1.

Pruning 1: The partition $[k+1, i]$ should be no larger than the last partition in $V[k][j-1]$. Based on **Corollary 4.2.2**, the last (smallest) partition for the $(k, j-1)$ pair is at most $\left\lfloor \frac{k}{j-1} \right\rfloor$. Since the next partition $[k+1, i]$ has $i-k$ records, we have:

$$i - k \leq \left\lfloor \frac{k}{j-1} \right\rfloor \leq \frac{k}{j-1} \Rightarrow k \geq i \cdot (1 - \frac{1}{j})$$

Pruning 2: The partition $[k+1, i]$ should be no less than the largest partition in the rest of the elements. Suppose that we have obtained the optimal $V[i][j]$. Recall that, we still need to assign the rest of the elements $n-i$ into $m-j$ partitions. In the optimal solution of $(n-i, m-j)$, the largest partition is at least $\left\lfloor \frac{n-i-1}{m-j} \right\rfloor + 1$. And thus we have,

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

Analysis. After changing the range of the third loop into $\lceil \max(i - \lfloor \frac{n-i-1}{m-j} \rfloor - 1, 1), i \cdot (1 - \frac{1}{j}) \rceil$. we can skip the calculation of $V[i][j]$ when $i \cdot (1 - \frac{1}{j}) < i - \lfloor \frac{n-i-1}{m-j} \rfloor - 1$, because we ensure that the cutting position at k is not required to compute $V[n][m]$. This pruning technique reduces the computation complexity to $O(n^2 \log m)$ (more details can be found in the appendix).

4.2.2 Step-based Pruning. The optimal value of $JC(P, CT)$ is largely affected by the ceiling function $\lceil \frac{\|\mathcal{P}\|}{c_R} \rceil$. In an extreme case, $\|\mathcal{P}\| = 1$ will have the same value $\lceil \frac{\|\mathcal{P}\|}{c_R} \rceil$ as when $\|\mathcal{P}\| = c_R = \lfloor b_R \cdot (B-2)/F \rfloor$. To address this, we prove **Theorem 4.3**.

Theorem 4.3. *There exists an optimal partitioning P_{opt} where the size of all partitions except the largest partition, is divisible by c_R.*

Note that **Theorem 4.3** and **Corollary 4.2.1** may give us two different partitioning strategies that have the same – optimal – cost. In other words, an optimal solution might satisfy either one of the two or both. For example, consider $n = c_R \cdot (B-2) + r$ $(0 < r < c_R)$. According to Corollary 4.2.1, we will group the last r records to form the smallest partition and the rest of records form $B-2$ partitions. However, Theorem 4.2 shows that we need to group the first r records together. Even though we may end up with different partitioning schemes, the optimal cost remains the same, due to Theorems 4.2 and 4.3, and Corollary 4.2.1. We now present a more relaxed version of 4.2.1 use to prove Theorem 4.3.

Corollary 4.3.1. *There exists an optimal partitioning scheme P_{opt} where all the partitions contain consecutive elements and the sequence $\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \rceil$ is in decreasing order.*

Proof. We use a similar proof for Corollary 4.2.1. Suppose that we start with the optimal partitioning scheme derived from **Theorem 4.2**. Suppose that we find two adjacent partitions that $\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \rceil$ are not in the non-decreasing order, as shown in Figure 8. We can apply **Lemma 4.1** multiple times until $\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \rceil$ of two adjacent partitions following the decreasing order. We can repeat this process for every two adjacent partitions until all the partitions are decreasingly sorted $\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \rceil$. Note that, we do not necessarily need to ensure $\|\mathcal{N}(i)|$ follows the decreasing order, which means that we may have $\|\mathcal{N}(i)\| < \|\mathcal{N}(i+1)\|$ but this is permitted as long as $\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \rceil = \lceil \frac{\|\mathcal{N}(i+1)\|}{c_R} \rceil$. Similarly, we can ensure that the join cost does not increase using Lemma 4.1 and thus the final partitioning scheme is still optimal. Proof completes. □

Based on the optimal solution P_{opt} from Corollary 4.3.1, we can prove Theorem 4.3 as follows.

Proof. We aim to find a process that can obtain another optimal solution which satisfies both Theorem 4.3 and Corollary 4.3.1. We note $\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_m$ as the partition sequence obtained from Corollary 4.3.1. As such, these partitions are sorted by $\lceil \frac{\|\mathcal{P}_j\|}{c_R} \rceil$ in a descending order. Now we select the largest $j (j \in [m])$ where $\|\mathcal{P}_j\|$ is not divisible by c_R. If $j = 1$, we do nothing as the partitioning scheme already satisfies Theorem 4.3. Otherwise, we move adjacent

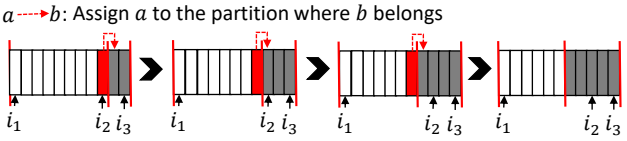


Figure 9: When $\|\mathcal{P}_j\|$ is not divisible by c_R, we can move elements from $\mathcal{P}_{j-1}$ to $\mathcal{P}_j$ until $\lceil \frac{\|\mathcal{P}_{j-1}\|}{c_R} \rceil < \lceil \frac{\|\mathcal{P}_j\|}{c_R} \rceil$ or $\|\mathcal{P}_j\|$ is divisible by c_R. In the above example, we note the partition containing $[i_2 + 1, i_3]$ as $\mathcal{P}_j$ and the previous partition containing $[i_1, i_2]$ as $\mathcal{P}_{j-1}$. Assuming $c_R = 5$, the above process stops when $\|\mathcal{P}_j\| = k \cdot 5)$ where $k \in \mathbf{N}$.

elements from $\mathcal{P}_{j-1}$ to $\mathcal{P}_j$ until $\lceil \frac{\|\mathcal{P}_{j-1}\|}{c_R} \rceil < \lceil \frac{\|\mathcal{P}_j\|}{c_R} \rceil$ or $\|\mathcal{P}_j\|$ is a multiple of c_R. When $\lceil \frac{\|\mathcal{P}_{j-1}\|}{c_R} \rceil < \lceil \frac{\|\mathcal{P}_j\|}{c_R} \rceil$, we swap elements according to the process mentioned in the proof of Corollary 4.2.1 so that the decreasing order of $\lceil \frac{\|\mathcal{P}_j\|}{c_R} \rceil$ is maintained. In other words, we obtain two consecutive partitions $\mathcal{P}_{j-1}^{new}, \mathcal{P}_j^{new}$ and we have $\|\mathcal{P}_{j-1}^{new}\| = \|\mathcal{P}_j^{old}\|, \|\mathcal{P}_j^{new}\| = \|\mathcal{P}_{j-1}^{old}\|$. We can repeat this process until the largest j of $\|\mathcal{P}_j\|$ can be divisible by c_R. Note that, when $\lceil \frac{\|\mathcal{P}_{j-1}\|}{c_R} \rceil < \lceil \frac{\|\mathcal{P}_j\|}{c_R} \rceil$, after we swap elements between $\mathcal{P}_{j-1}$ and $\mathcal{P}_j$, $\mathcal{P}_{j-1}$ will be selected to add elements from $\mathcal{P}_{j-2}$. The above process already ensures that the final partition contains consecutive elements and the descending order elements based on their CT value. In this process, when we move elements to $\mathcal{P}_j$, $\lceil \frac{\|\mathcal{P}_j\|}{c_R} \rceil$ does not change since $\|\mathcal{P}_j\|$ is not divisible by c_R. Therefore, after we move one element i to $\mathcal{P}_j$, all the elements from $\mathcal{P}_j$ contribute the same cost and all the elements (except i) from $\mathcal{P}_{j-1}$ have equivalent or even lower cost since $\|\mathcal{P}_{j-1}\|$ decreases. For element i, the cost cannot increase since we maintain the descending order of $\lceil \frac{\|\mathcal{P}_j\|}{c_R} \rceil$. Specifically, when we move element i from $\mathcal{P}_{j-1}$ to $\mathcal{P}_j$, $\lceil \frac{\|\mathcal{N}(i)\|}{c_R} \rceil$ must decrease or remain the same. Proof completes. □

Applying **Theorem 4.3** in Algorithm 1 works as follows. Initially, we put the first $n \mod c_R$ records in the first partition. And then we can change the step size (line 7 and line 10) of the MatrixDP algorithm to c_R. For the *Cut* matrix, the number of rows can be changed into $\lceil \frac{n}{c_R} \rceil + 1$. Recall that Theorem 4.3 is based on Corollary 4.3.1. We can apply the monotonicity-based pruning for Corollary 4.3.1 to avoid conflicts with Corollary 4.2.1. To be specific, instead of checking $\|\mathcal{P}_{j-1}\| \geq \|\mathcal{P}_j\|$, we examine if $\|\mathcal{P}_{j-1}\| \geq \|\mathcal{P}_j\| + c_R$. As now we change the step size to c_R, we will just need two more constant comparisons to calculate the lower bound and the upper bound of the current partition size. The optimal partitioning scheme obtained by Theorem 4.3 naturally follows an approximate rounded hash scheme (discussed in Section 3) since the size of each partition is finely controlled and there is only one partition of which the size is not divisible by c_R.

Analysis. Recall that m can as large as $B-1$, F is a constant close to 1, and that $c_R = \lfloor b_R \cdot (B-2)/F \rfloor = \Omega(m)$. Therefore, the time

complexity is reduced to $O\left(\left(\frac{n}{\lfloor b_R \cdot (B-2)/F \rfloor}\right)^2 \log m\right) = O\left(\frac{n^2 \log m}{m^2}\right)$ and the space complexity becomes $O\left(\frac{n}{\lfloor b_R \cdot (B-2)/F \rfloor} \cdot m\right) = O(n)$.

4.3 Enforcing Memory Constraints

In the algorithms that calculate the exact optimal partitioning, we need to maintain the mapping for each row to its corresponding partition. This metadata may take up too much space, and eventually violate the memory constraints we have to respect. For example, assuming a hash map is used to store the assignment for each key, when we have 2 million 4-byte keys from R, we need at least 8 MB space or even more due to the fudge factor. Besides, we may not even have the whole CT since maintaining such a correlation table would also incur overhead. In this section, we propose an approximation solution to address the above issues and maintain memory consumption bounded.

4.3.1 Divide-and-Partition. Inspired by the MCV (Most Common Variables) optimization [27] in PostgreSQL, we assume that only the top-k frequently matching records are tracked by the database system. We define $SubCT_k = \{CT[i] | i \in [n-k+1, n]\}$ and $SubKeys_k = \{key_i | i \in [n-k+1, n]\}$ where key_i means the corresponding key for $CT[i]$. In terms of the memory footprint, $SubCT_k$ and $SubKeys_k$ can take up at most $k \cdot (key\_size + 4)$ bytes assuming $CT[i]$ is a 4-byte unsigned integer. Next, we use $SubCT_k$ and $SubKeys_k$ to produce a hash map (noted by HM, and its memory footprint, in pages, is noted by B_{HM}), that records the partitioning assignment between each key from $SubKeys_k$ and its partition id $j \in [m_k]$, where m_k represents the number of partitions assigned to $SubKeys_k$. Note that, based on whether a key belongs to HM, we logically divide the key domain into two subsets and further divide the partition space into two groups $\{\mathcal{P}_{j_1} | j_1 \in [m_k]\}, \{\mathcal{P}_{j_2} | j_2 \in [m_k+1, m-B_{HM}]\}$. For each key that is *not* in $SubKeys_k$, we can assign it to a random partition in $\{\mathcal{P}_j | j \in [m_k+1, m-B_{HM}]\}$. We define $m_r = m - B_{HM} - m_k$, and then we break down the memory budget into four components, as shown in Figure 10. The divide-and-partitioning process is summarized in Algorithm 3.

Algorithm 3 $DivideAndPartition(HM, m_k, R)$

```
for every block blk_i from relation R do
    for every record r = (key, value) from block blk_i do
        if key ∈ HM then
            PartID ← HM[key] (HM[key] ∈ [m_k])
        else
            h ← hash(key) mod α · ⌈(n−k)/c_R⌉   ▷ Rounded Hash
            PartID ← (h mod (m − m_k − B_HM)) + m_k + 1
        end if
        assign r to partition P_PartID
    end for
end for
```

4.3.2 How to Obtain a Good m_k. Algorithm 3 needs to know HM and m_k. Essentially, we need to determine the optimal number of partitions for $SubKeys_k$. Since the number of partitions cannot be larger than $B-1$, when more partitions are assigned to $SubKeys_k$, fewer partitions are available for the remaining keys, as seen in Figure 10. Assigning more partitions to a specific key set means that the size of each partition $\|\mathcal{P}_j\|$ is smaller and thus $\left\lceil \frac{\|\mathcal{P}_j\|}{c_R} \right\rceil$ could be smaller, which further leads to lower join cost. Therefore, we have to leverage the trade-off in terms of the join cost between $SubKeys_k$ and the remaining keys.

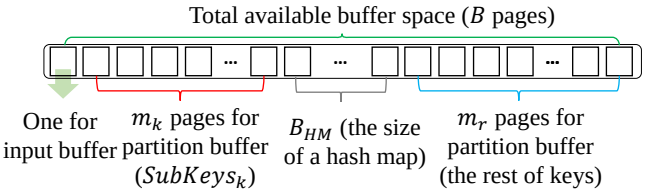


Figure 10: To ensure all the records of which the keys belong to $SubKeys_k$ can be assigned to the ideal partition, we have to maintain an in-memory hash map. As shown above, in addition to one page for input and $m_k + m_r$ pages for partitioning buffer, we have B_{HM} pages reserved to store the hash map.

To find out the best m_k, we can iterate over every possible m_k, calculate the overall join cost, and select the one with minimum cost. While we can derive the exact join cost of $SubKeys_k$ using the pruned Algorithm 1, we can only *estimate* the join cost of the remaining keys because we do not know how the hash function partitions the remaining keys. Suppose we have $m_r = m - m_k - B_{HM}$ partitions for the rest of keys, the join cost can be then modeled as follows:

$$JC(P_{Hash}, CT') = \sum_{j=m_k+1}^{m} \sum_{i \in \mathcal{P}_j} CT[i] \cdot \left\lceil \frac{\|\mathcal{P}_j\|}{c_R} \right\rceil$$

where P_{Hash} represents the partitioning matrix produced by the hash function and $CT' = CT - SubCT_k$, which is the sub-table consisting of the first $n-k$ values. Assigning records into a random partition can be treated as a balls-into-bins problem, and thus the size of each partition can be approximated by a Poisson distribution with $\lambda = \frac{n-k}{m_r}$, s.t., $\sum_{j=m_k+1}^{m} \|\mathcal{P}_j\| = n-k$. As each partition has the same expected size, the expected overall join cost is given as follows:

$$\begin{aligned} E[JC(P_{Hash}, CT')] &\approx \sum_{j=m_k+1}^{m} \sum_{i \in \mathcal{P}_j} CT[i] \cdot \left\lceil \frac{E[\|\mathcal{P}_j\|]}{c_R} \right\rceil \\ &= \sum_{j=m_k+1}^{m} \sum_{i \in \mathcal{P}_j} CT[i] \cdot \left\lceil \frac{n-k}{m_r \cdot c_R} \right\rceil = \left\lceil \frac{n-k}{m_r \cdot c_R} \right\rceil \cdot \left(\sum_{i=1}^{n-k} CT[i] \right) \end{aligned} \tag{13}$$

Note that $\sum_{i=1}^{n-k} CT[i]$ is essentially $PPS[n-k]$, hence, the above join cost only relies on the two input parameters m_r and k. We define $g_{\text{NBJ}}(m_r, k) = E[JC(P_{Hash}, CT)]$ since we assume NBJ is used to execute the partition-wise join in the above model. Similarly, we can also define $g_{\text{SMJ}}(m_r, k)$ and $g_{\text{GHJ}}(m_r, k)$ using the estimation equation in Table 1. We further define $g(m_r, k)$ as the minimum cost among $g_{\text{NBJ}}(m_r, k)$, $g_{\text{SMJ}}(m_r, k)$, and $g_{\text{GHJ}}(m_r, k)$. As such, the overall join cost can be estimated as the summation between $g(m_r, k)$ and the optimal cost obtained by MatrixDP$(SubCT_k, n-k, m_k)$.

Theoretically, $\lceil \log_2 m_k \rceil$ bits or $\lceil \frac{1}{8} \cdot \log_2 m_k \rceil$ bytes are sufficient to store the partition id. For simplicity, we use `uint16_t` (2 bytes) in our implementation. When $m_k = 1$, we build a hash set instead. Considering the fudge factor and the key size, the number of pages of the in-memory hash map is equal to:

$$B_{HM}(k, m_k) = \left\lceil \frac{k \cdot F \cdot (\text{key\_size} + \lceil \frac{1}{8} \cdot \log_2 m_k \rceil)}{\text{page\_size}} \right\rceil \quad (14)$$

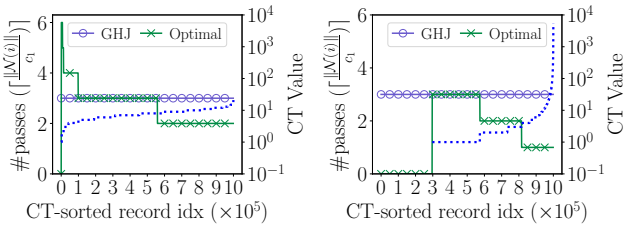
Although $SubKeys_k$ are tracked, we do not necessarily build the hash map for all keys from $SubKeys_k$. Instead, we find the best combination $k^{opt} \leq k$ and m_k^{opt} that can minimize the estimated join cost. The whole algorithm is shown in Algorithm 4. In lines $1-3$, if k is smaller than c_R, one partition is sufficient to achieve the optimal cost for $SubKeys_k$. In lines $9-18$, we iterate over all possible pairs (i, j) – i is the number of top matching keys and j is the number of partitions assigned to $SubKeys_i$ – and select the pair with the minimum expected cost. According to Theorem 4.3, the minimum partition size for $SubKeys_k$ is c_R, and thus the maximum m_k is no more than $\lceil k/c_R \rceil$. The expected cost is estimated by the summation between the optimal cost obtained from $V[k][m_k].cost$ and the expected cost of the remaining keys using $g(m_r, i)$ (line 12). When $m_k^{opt} = 1$, we can create a hash set for $SubKeys_k$ (lines $19-21$) to save memory. The overall cost of Algorithm 4 is driven by MatrixDP and the nested for-loop (lines $7-15$), and thus, the total complexity is $O(k^2 \cdot \log m/m^2 + k^2/m) = O(k^2/m)$. At the same time, we have to limit k so that $B_{HM}(k, m_k) + m_k$ can fit in memory. Since $m_k \leq \lceil k/c_R \rceil$, we have:

$$B_{HM}(k, m_k) + m_k \leq B_{HM}(k, \lceil k/c_R \rceil) + \lceil k/c_R \rceil \leq B - 2 \quad (15)$$

We can use the largest k (noted by k_{max}) that satisfies Equation (15) to replace k if $k > k_{max}$.

Algorithm 4 $FindBestMk(SubCT_k, SubKeys_k, k, n, m)$

```
1:  if k < c_R then
2:      Create a hash set HS for all keys from SubKeys_k
3:  else
4:      m_k^max ← ⌈k / c_R⌉
5:      V ← MatrixDP(SubCT_k, k, m_k^max)
6:      m_k^opt ← 0; k^opt ← 0; cost^opt ← MAX_INT
7:      for i ← 1 to k do
8:          for j ← 1 to ⌈i / c_R⌉ do
9:              m_r ← m − B_HM(i, j) − j
10:             cost^tmp ← V[i][j].cost + g(m_r, i)
11:             if cost^opt < cost^tmp then
12:                 m_k^opt ← j; k^opt ← i; cost^opt ← cost^tmp
13:             end if
14:         end for
15:     end for
16:     if m_k^opt == 1 then                    ▷ Create a hash set
17:         Build HS for SubKeys_{k^opt}
18:     else                                    ▷ Create a hash map
19:         CutPositions ← Get_Cutting_Pos(V, k^opt, m_{k^opt})
20:         Build HM based on CutPositions and SubKeys_{k^opt}
21:     end if
22: end if
```



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

Figure 11: GHJ uniformly partitions and thus the number of required passes ($\lceil \|\mathcal{N}(i)\|/c_R \rceil$) for each partition is the same, regardless of the correlation. In contrast, using our optimal partitioning algorithm, the number of required passes follows a non-monotonically decreasing trend as CT values increase (excluding records with $CT[i] = 0$).

4.4 Example

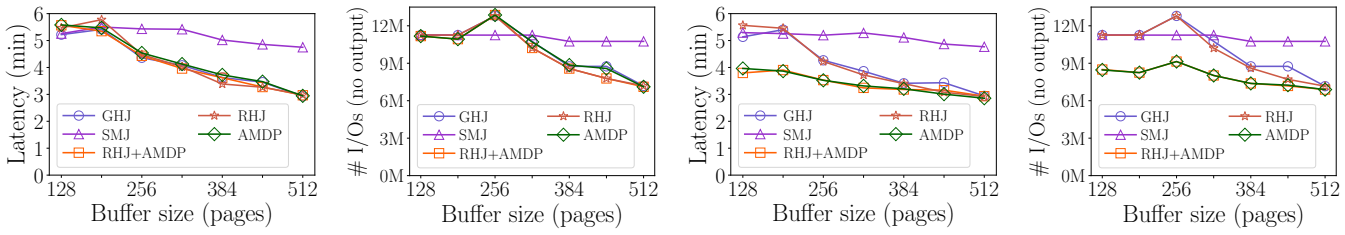
To illustrate the optimal partitioning scheme, we conduct a case study to visualize the difference between the optimal scheme and the state-of-the-art that performs uniform partitioning (GHJ). Suppose we join relation R with S, with $n_R = 1$M, $n_S = 8$M keys and the join key being the primary key in R and the foreign key in S. Assuming 4-byte entries for R, the relation occupies $250K$ pages. The available buffers are 320 pages. Since $320 < \sqrt{250K \cdot F}$, the join is executed using GHJ. As shown in Figure 11a and 11b, after applying the random partitioning in GHJ, the partition size is around $250K/319 \approx 784$ pages which requires $\lceil 783/(318 \times F) \rceil = 3$ passes to scan every partition from S. In contrast, the optimal partitioning scheme in Figure 11a allows the number of passes to vary from 2 to 6. Those records with low CT values have more passes (e.g., 4,5,6) and records with high CT values have fewer passes (e.g., 2), which is consistent with corollary 4.3.1. This behavior is more prominent for more skewed join attribute correlation, as shown in Figure 11b (the optimal partitioning naturally excludes values that do not have any match and thus the partition size is 0 for non-matching records).

5 EXPERIMENTAL ANALYSIS

We now present our experimental results for the methods introduced in Sections 2, 3, and 4. We use the same experimental setup as described in Section 2.

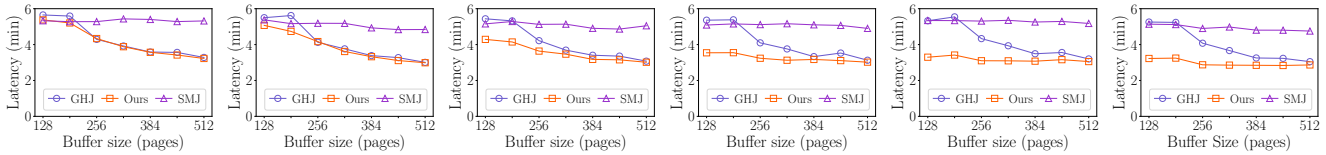
Approaches Compared. We use as baselines GHJ(Opt) that extends grace hash join (introduced in §2) and sort-merge join (SMJ). We compare those approaches with Rounded Hash Join (RHJ for short, introduced in §3), Approximate MatrixDP (AMDP, introduced in §4.3), and the combined version RHJ+AMDP. The optimal partitioning method (§4.1) is not used for latency experiments since it ignores the memory constraint due to the large hash map for the partitioning assignment. By default, we assume the light join optimizer mentioned in Section 2 is always applied to RHJ and ADMP to ensure a fair comparison. As such, we further simplify the notation GHJ(Opt) to GHJ in the remainder of the section. For AMDP, we assume that the top 5% frequently matching keys are tracked and thus $k = 50K$ when $n = 1M$.

Implementation All approaches are implemented in C++ (compiled with cmake 3.20.2 and gcc 10.1.0) and we use a CentOS Linux



(a) Uniform (Latency) (b) Uniform (# I/Os) (c) Zipfian (Latency) (d) Zipfian (# I/Os)

Figure 12: When the workload is uniform, RHJ and the combined method are slightly better than GHJ. In contrast, when the workload is skewed, AMDP and the combined method are significantly better than GHJ and SMJ.



(a) Zipfian ($\alpha = 0.5$) (b) Zipfian ($\alpha = 0.7$) (c) Zipfian ($\alpha = 0.9$) (d) Zipfian ($\alpha = 1.1$) (e) Zipfian ($\alpha = 1.3$) (f) Zipfian ($\alpha = 1.5$)

Figure 13: Our approach yields comparable performance to GHJ when the workload is less skewed and we gain more benefit as we increase the workload skewness. Since the minimum I/O cost in partitioned join is lower bounded by $3 \cdot (\|R\| + \|S\|)$, we cannot get further improvement when the latency is close to the lower bound ($\alpha = 1.3$ and $\alpha = 1.5$).

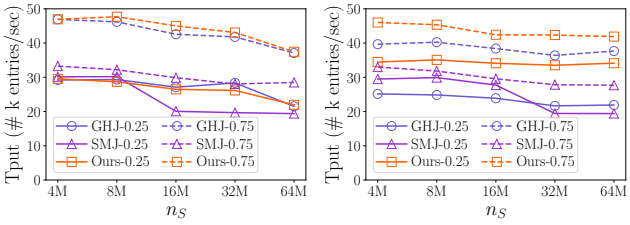
release 8 with XFS and Linux kernel 4.18.0. Our code is available at: https://github.com/BU-DiSC/Partition-For-Equi-Join.

5.1 Experimental Results

RHJ+AMDP Dominates. We first conduct a sensitivity analysis to see how much performance improvement each component contributes, compared to GHJ and SMJ. As shown in Figure 12a, when the buffer size is smaller than 256 pages, all the methods have similar performance since they have similar expected #I/Os ($5 \cdot (\|R\| + \|S\|)$). This is because in this case, the cost of GHJ and SMJ is driven by having to do one more pass over the data (to partition or sort accordingly), leading in both cases to $2 \cdot (\|R\| + \|S\|)$ extra I/Os. When we have a larger buffer size, the latency of SMJ does not change much as the number of required passes does not decrease. In this case, partitioned-based methods have smaller partitions and thus only NBJ is applied to execute partition-wise join instead of SMJ or GHJ. Note that AMDP has almost the same performance as GHJ under a uniform workload, because AMDP exactly performs GHJ/RHJ for keys with low CT value. Even though $SubKeys_{kopt}$ are prioritized in AMDP, these keys have similar average CT value to the average CT value and thus AMDP yields similar #I/Os compared to GHJ under a uniform workload, as shown in Figure 12b. From this figure, we can also observe that RHJ and the combined method generally have lower #I/Os compared to other methods, which is not reflected in Figure 12a. Since the write latency is 4.5× more expensive than the read latency in our device, the overall latency is dominated by the partitioning phase, which explains that #I/Os reduction does not yield observable latency difference. When the workload becomes skewed, AMDP and RHJ+AMDP yield much better performance compared to GHJ, as shown in Figure 12c. Since the workload is skewed, most records in S are matched with $SubKeys_{kopt}$ from R, and they are assigned into a small partition in AMDP (only one pass is required to scan them, in contrast to three passes for most records for uniform correlation). Besides, when the buffer size is 256-page, we have more #I/Os in Figures 12b and 12d but lower latency in Figures 12a and 12c, which comes from the read/write asymmetry as explained in Section 2.

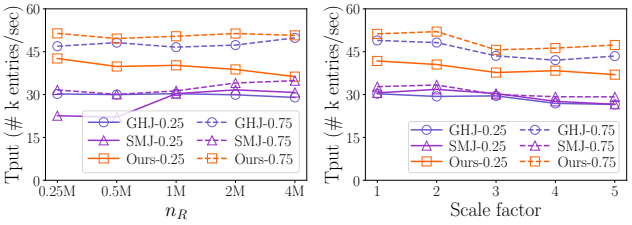
Benefits Are Pronounced for Higher Skew. As the join cost in Equation (9) is largely affected by the correlation table CT, we then evaluate the effect of various correlation skewness. We conduct our experiments using 6 different Zipfian correlations with varying the Zipfian parameter, α, from 0.5 to 1.5. As the combined RHJ and AMDP method can always outperform both RHJ and AMDP, we always apply the combined version (abbreviated by **ours**) in the rest of the experiments. As shown in Figure 13a, our approach is slightly better than GHJ and has comparable performance to SMJ when the number of buffer pages is smaller than 256. When we have more buffer pages, our approach has a similar performance to GHJ due to the close-to-uniform correlation. When we increase α from 0.5 to 1.1 (increasing the correlation skew), the latency of our approach gradually drops until it is close to the latency of a GHJ with the ideal memory available to it (~3 minutes). After that, increasing α does not further reduce latency since we are already at the lower bound of what a partitioned join can achieve.

RHJ+AMDP Scales With Data Size. We now conduct a scalability experiment to examine the join throughput, defined as $\frac{n_R+n_S}{total\_time}$. In this experiment, we first fix n_R and vary n_S from 4 million to 64 million. Since n_R is fixed, the buffer size does not change as n_S increases. We use two distinct memory budgets: $\sqrt{\|R\| \cdot F}/4$ and $3 \cdot \sqrt{\|R\| \cdot F}/4$. We note the algorithm with buffer size $\sqrt{\|R\| \cdot F} \cdot \beta$ by algorithm-β (e.g., GHJ-0.25 means applying GHJ using $\sqrt{\|R\| \cdot F}/4$ pages). As we can see from Figure 14a, the join throughput gradually drops when we increase n_S, and the difference between our method and GHJ is negligible when the correlation is uniform. When it comes to a skewed distribution, as shown in Figure 14b, the throughput becomes even more stable. Specifically, when $n_S = 64M$, the



(a) Uniform (b) Zipfian

Figure 14: For a fixed R, as the size of S increases, the throughput decreases due to insufficient memory budget, however, our approach still outperform GHJ and SMJ.



(a) Varying $\|R\|$ and fixing $\|S\|$ (b) Varying both proportionally

Figure 15: For a Zipfian join key correlation, our approach dominates under limited memory budget, regardless of the input data size.

throughput under uniform correlation even becomes smaller than the one under Zipfian correlation. We then examine the I/O count for these two cases and the difference is negligible. Ideally, the throughput should remain nearly unchanged. In fact, a few partitions can be very large and thus the original partition-wise join method SMJ is replaced by recursive GHJ since the limited memory cannot hold the larger partition. While both of two algorithms can achieve the minimum I/O cost, SMJ is generally faster than GHJ due to the read/write asymmetry. Since our method and GHJ still use a hash function to distribute data, it is still possible that a few partitions might be a bit larger than the threshold which triggers GHJ. For a skewed correlation, since the majority of records from n_S match only a few records from n_R, which are pre-partitioned by our approximate algorithm, SMJ is more frequently picked and thus yields more stable throughput.

Next, we do two more experiments: 1) Vary the size of both R and S proportionally, 2) Keep S fixed and vary R. For these two experiments, we employ only the Zipfian correlation. Note that, as we now vary n_R and thus $\|R\|$ changes, hence, k and the buffer size should also change proportionally to adapt to different $\|R\|$. Similarly, we still use just two distinct memory budgets ($\sqrt{\|R\| \cdot F}/4$, $3 \cdot \sqrt{\|R\| \cdot F}/4$). To be specific, in the first experiment, we vary the whole data size from 9GB (when the scale factor is 1, we have 1GB for R and 8GB for S) to 45GB (when the scale factor is 5, we have 5GB for R and 40GB for S), and maintain the average number of matching keys to 8. In the second experiment, we vary n_R from 0.25 million to 4 million. As shown in Figure 15a, when we vary n_R, SMJ's throughput increases because the memory budget is proportional to $\sqrt{\|R\|}$ and having a larger buffer allows sorting S in fewer passes. Contrary to SMJ and GHJ, our approach has stable throughput. Note that, even though SMJ has higher throughput with larger n_R, SMJ still cannot beat GHJ nor our approach with the same memory budget. On the other hand, when we vary the size of both relations proportionally (including the buffer size), the after scale factor 3, as we can see from Figure 15b. This happens for the same reason mentioned earlier: recursive GHJ replaces SMJ for larger data sizes. Overall, with the scalability experiments, we observe that the benefits of our approach do not depend on the data size. Notice if we vary the size of only one relation or vary the size of both relations, we see our approach providing benefits independent on the data size.

6 RELATED WORK

Classical Joins and Optimizations. Traditional join algorithms include Nested-Block Join (NBJ), Grace Hash Join (GHJ), Simple Hash Join (SHJ), and Sort-Merge Join (SMJ). GHJ is preferable when the available memory is small [16]. When we have larger memory, Hybrid Hash Join (HHJ), and many other optimizations [19, 23, 24, 34] are proposed to reduce the I/O cost. One optimization [11] suggests caching frequently matched keys to save #I/Os, which can be perceived as a special case of Corollary 4.2.1, where the partition size of a *cached* record is zero. However, HHJ normally assumes the memory budget is larger than $\sqrt{F \cdot \|R\|}$ and the extra memory is used to store one or more partitions. As such, it is not applicable when the memory is smaller than $\sqrt{F \cdot \|R\|}$.

In-Memory Joins. When the memory is enough to store the entire relations, many in-memory join algorithms can be applied [4, 20, 29]. Similar to disk-based joins, most 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 [2]) and non-partitioned joins (e.g., CHT [5]). In-memory joins are optimized for memory access, parallelism, cache/TLB (Translation Lookaside Buffer) misses, and specialized CPU instructions. Although disk-based joins focus on #I/Os, when two partitions both fit in memory after the partitioning phase, we can still benefit from an optimized in-memory join algorithm.

Other Partitioning Problems in Database Systems. Generally, due to the exponential search space, many partitioning problems in the database field have been proven to be NP-Hard. For example, data partitioning is also used to reduce #I/Os when answering queries, which can be built from predicates of historical queries. However, finding the optimal data partitioning is proved to be an NP-Hard problem [31]. A deep learning framework WOODBLOCK is proposed to approximate the optimal solution [33]. To skip unmatched records before a join, data partitioning can also be built through MTO [14]. Another partitioning example can be found when we try to put variable-size records into fixed-size pages. Maximizing space utilization is also claimed as the typical *online object placement* problem [17], which is a well-known NP-Hard problem. Besides, when the join is executed in a distributed environment, data partitioning is applied to reduce network transmission. This optimization problem can be written in a form of integer linear programming, which has been proved to be NP-Hard [21].

7 CONCLUSIONS

In this paper, we observe that the state-of-the-art disk-based GHJ yields sub-optimal performance under constrained memory. We address the limitations of GHJ by (i) extensive use of NBJ (ii) Rounded Hash Join, and (iii) correlation-aware optimal join partitioning. We further design an approximate partitioning algorithm based on top-k frequently matched keys to enforce the memory constraint. Our results show that our proposed algorithm outperforms GHJ by 1.8× under a limited memory budget and the performance gain becomes more prominent under a skewed workload.

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

[3] 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

[4] 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

[5] 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

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

[7] 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

[8] 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

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

[10] Cisco. 2018. Cisco Global Cloud Index: Forecast and Methodology, 2016–2021. *White Paper* (2018). https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html{#}{_}Toc503317525

[11] 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

[12] 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

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

[14] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 418–431. https://doi.org/10.1145/3448016.3457270

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

[16] 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

[17] Shiva Jahangiri, Michael J Carey, and Johann-Christoph Freytag. 2021. Design Trade-offs for a Robust Dynamic Hybrid Hash Join (Extended Version). *CoRR* abs/2112.0 (2021). https://arxiv.org/abs/2112.02480

[18] 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

[19] 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

[20] 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

[21] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. 2017. AdaptDB: Adaptive Partitioning for Distributed Joins. *Proceedings of the VLDB Endowment* 10, 5 (2017), 589–600. http://www.vldb.org/pvldb/vol10/p589-lu.pdf

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

[23] 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

[24] Hwee Hwa Pang, Michael J Carey, and Miron Livny. 1993. Partially preemptible hash joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 59–68.

[25] 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)*.

[26] 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)*.

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

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

[29] 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

[30] 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

[31] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained Partitioning for Aggressive Data Skipping. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1115–1126. https://doi.org/10.1145/2588555.2610515

[32] 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

[33] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 193–208. https://doi.org/10.1145/3318464.3389770

[34] Hansjörg Zeller and Jim Gray. 1990. An Adaptive Hash Join Algorithm for Multiuser Environments. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 186–197. http://www.vldb.org/conf/1990/P186.PDF

Appendices

A COMPLEXITY ANALYSIS FOR MONOTONICITY-BASED PRUINING

Based on two pruning inequalities (i.e., **Pruning 1** and **Pruning 2**) from corollary 4.2.2, the range for k is at most $\frac{i}{j} - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1$. If $\frac{i}{j} - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1 > \frac{i}{j}$, we can even skip the specific j (line 8 in Algorithm 1) because we do not have to iterate any k in this case. Therefore, the complexity for the third loop (line 8 $\sim$ 16) is

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

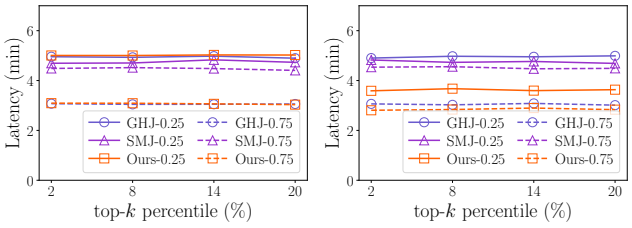
. Summing up the complexity for all possible i and j, we have:

$$\begin{aligned}
&\sum_{i=1}^{n}\sum_{j=2}^{m-1} \max(\frac{i}{j} - \left\lfloor \frac{n-i-1}{m-j} \right\rfloor - 1, 0) \\
&\leq \sum_{i=1}^{n}\sum_{j=2}^{m-1} \max(\frac{i}{j} - \frac{n-i}{m-j}, 0) = \sum_{j=2}^{m-1}\sum_{i=\frac{j\cdot n}{m}}^{n} \frac{i\cdot m - n\cdot j}{j\cdot(m-j)} \\
&= \sum_{j=2}^{m-1} \frac{\frac{n\cdot j + n\cdot m}{2}\cdot\left(n - \frac{n\cdot j}{m} + 1\right) - n\cdot j\left(n - \frac{n\cdot j}{m} + 1\right)}{j\cdot(m-j)} \\
&= \sum_{j=2}^{m-1} \frac{\frac{n\cdot(m-j)}{2}\cdot\left(n - \frac{n\cdot j}{m} + 1\right)}{j\cdot(m-j)} = \sum_{j=2}^{m-1} \frac{n\cdot\left(n - \frac{n\cdot j}{m} + 1\right)}{2j} \\
&= \frac{1}{2}\left(\sum_{j=2}^{m-1} n^2\cdot\left(\frac{1}{j} - \frac{1}{m}\right) + \sum_{j=2}^{m-1}\frac{n}{j}\right) = O(n^2\log m)
\end{aligned} \tag{16}$$

As shown in Equation (16), monotonicity-based pruning reduces the time complexity to $O(n^2 \log m)$

B MICRO-BENCHMARK

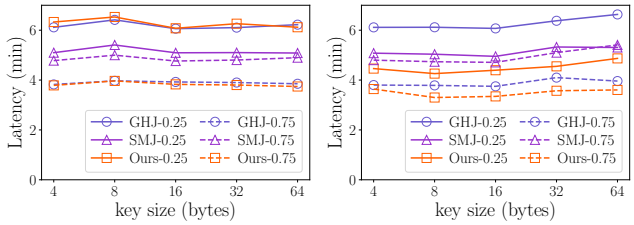
Varying k. In this experiment, we examine the effect of k using 2 memory budget ($\sqrt{\|R\| \cdot F}/4$ and $3 \cdot \sqrt{\|R\| \cdot F}/4$ pages) by varying k from $20K$ to $200K$. As mentioned earlier, tracking top-k frequently matching keys does not necessarily mean we use top-k elements to build the hash map/set in our approximation algorithm. As shown in Figure 16a, all the orange lines do not even change when k grows. Larger k does not yield significant performance improvement.



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

Figure 16: Varying k.

Varying key size. We further evaluate the impact of the key size since our approaches relies on the in-memory hash map to enforce the partitioning assignment. In this micro-benchmark. we use the same memory budget as before and vary the key size from 4 bytes to 64 bytes. Observe from Figure 17a and 17b, in most cases, the key size does not have huge impact over the join performance. Although larger key size indicates higher CPU complexity, this difference is negligible when the performance is dominated by #I/Os.



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

Figure 17: Varying key size.