Near-Optimal Distributed Band-Joins through Recursive Partitioning

Rundong Li∗
Google, USA
eadan.lee@gmail.com
Wolfgang Gatterbauer
Northeastern University, USA
w.gatterbauer@northeastern.edu
Mirek Riedewald
Northeastern University, USA
m.riedewald@northeastern.edu

ABSTRACT
We consider running-time optimization for band-joins in a distributed system, e.g., the cloud. To balance load across worker machines, input has to be partitioned, which causes duplication. We explore how to resolve this tension between maximum load per worker and input duplication for band-joins between two relations. Previous work suffered from high optimization cost or considered partitionings that were too restricted (resulting in suboptimal join performance). Our main insight is that recursive partitioning of the join-attribute space with the appropriate split scoring measure can achieve both low optimization cost and low join cost. It is the first approach that is not only effective for one-dimensional band-joins but also for joins on multiple attributes. Experiments indicate that our method is able to find partitionings that are within 10% of the lower bound for both maximum load per worker and input duplication for a broad range of settings, significantly improving over previous work.

CCS CONCEPTS
• Information systems → MapReduce-based systems; Relational parallel and distributed DBMSs.

KEYWORDS
band-join; distributed joins; running-time optimization

ACM Reference Format:

∗Work performed while PhD student at Northeastern University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD’20, June 14–19, 2020, Portland, OR, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6373-6/20/06...$15.00
https://doi.org/10.1145/3318464.3389750

1 INTRODUCTION
Given two relations S and T, the band-join S ≻ρ T returns all pairs (s ∈ S, t ∈ T) that are “close” to each other. Closeness is determined based on band-width constraints on the join attributes which we also call dimensions. This is related to (but in some aspects more general than, and in others a special case of) similarity joins (see Section 3). The Oracle Database SQL Language Reference guide [23] presents a one-dimensional (1D) example of a query finding employees whose salaries differ by at most $100. Their discussion of band-join specific optimizations highlights the operator’s importance. Zhao et al [45] describe an astronomy application where celestial objects are matched using band conditions on time and coordinates ra (right ascension) and dec (declination). This type of approximate matching based on space and time is very common in practice and leads to three-dimensional (3D) band-joins like this:

EXAMPLE 1. Consider bird-observation table B with columns longitude, latitude, time, species, count, and weather table W, reporting precipitation and temperature for location-time combinations. A scientist studying how weather affects bird sightings wants to join these tables on attributes longitude, latitude, and time. Since weather reports do not cover the exact time and location of the bird sighting, she uses a band-join to link each bird report with weather data for “nearby” time and location, e.g., |B.longitude − W.longitude| ≤ 0.5 AND |B.latitude − W.latitude| ≤ 0.5 AND |B.time − W.time| ≤ 10.
We are interested in minimizing end-to-end running time of distributed band-joins, which is the sum of (1) optimization time (for finding a good execution strategy) and (2) join time (for the join execution). Join time depends on the data partitioning used to assign input records to worker machines. As seen in Figure 1, band-joins generalize equi-join and Cartesian product. Partitioning algorithms with optimality guarantees are known only for these two extremes [1, 26, 29].

**Example 2.** To see why distributed band-joins are difficult, consider a 1D join with band width $\varepsilon = 1$ of $S = \{1, 2, 3, 5, 6, 8, 9, 10\}$ and $T = \{1, 5, 6, 10\}$ on $w = 2$ workers. For balancing load, we may split $S$ on value $x$ and send the left half to worker $w_1$ and the right half to $w_2$ (see Figure 2a). To not miss results near the split point, all $T$-tuples within band width $\varepsilon = 1$ of $x$ have to be copied across the boundary. Figure 2b shows the resulting input and output tuples on each worker, with duplicates in orange. By splitting in sparse regions of $T$, e.g., on $y_1$ and $y_2$ (Figures 2c and 2d), perfect load balance can be achieved without input duplication.

The main contribution of this work is a novel algorithm RecPart (Recursive Partitioning) that quickly and efficiently finds split points such as $y_1$ and $y_2$. To do so, it has to carefully navigate the tradeoff between load balance and input duplication. For instance, by itself appears like a poor choice from a load-balance point of view. It takes the additional split on $y_2$ to unlock $y_1$’s true potential.

**Overview of the approach.** We propose recursive (i.e., hierarchical) partitioning of the join-attribute space, because it offers a broad variety of partitioning options that can be explored efficiently. As illustrated by the split tree in Figure 3, each path from the root node to a leaf defines a partition of the join-attribute space as a conjunction of the split predicates in the nodes along the path. Like decision trees in machine learning [16], RecPart’s split tree is grown from the root, each time recursively splitting some leaf node. This step-wise expansion is a perfect solution for the problem of navigating two optimization goals: minimizing max worker load and minimizing input duplication. As RecPart grows the tree, input duplication is monotonically increasing, because more tuples may have to be copied across a newly added split boundary. At the same time, large partitions are broken up and hence load balance may improve.

To find good partitionings, it is important to (1) use an appropriate scoring function to pick a good split point (e.g., choose $y_1$ or $y_2$ over $x$ in Figure 2) and to (2) choose the best leaf to be split next. We propose the ratio between load balance improvement and additional input duplication for both decisions. In Example 2, this would favor $y_1$ and $y_2$ over $x$, because they add zero duplication. Similarly, a leaf node with a zero-duplicate split option would be preferred over a leaf whose split would cause duplication.

When a leaf becomes so small that virtually all tuples in the corresponding partition join with each other, then it is not split any further. However, if the load induced by that partition is high, then the leaf “internally” uses a grid-style partitioning inspired by 1-Bucket [29] to create more fine-grained partitions. This is motivated by the observation that the band-join computation in a sufficiently small partition behaves like a Cartesian product—for which 1-Bucket was shown to be near-optimal.

**Main contributions.** (1) We demonstrate analytically and empirically that previous work falls short either due to high optimization time (to find a partitioning) or due to high join time (caused by an inferior partitioning), especially for band-joins in more than one dimension.

(2) To address those shortcomings, we propose recursive partitioning of the multidimensional join-attribute space. Given a fixed-size input and output sample, our algorithm
RecPart finds a partitioning in \(O(w \log w + wd)\), where \(w\) is the number of workers and \(d\) is the number of join attributes, i.e., the dimensionality of the band-join. RecPart is inspired by decision trees [16], which had not been explored in the context of optimizing running time of distributed band-joins. To make them work for our problem, we identify a new scoring measure to determine the best split points: ratio of load variance reduction to input duplication increase. It is informed by our observation that a good split should improve load balance with minimal additional input duplication. We also identify a new stopping condition for tree growth.

(3) While we could not prove near-optimality of RecPart’s partitioning, our experiments provide strong empirical evidence. Across a variety of datasets, cluster sizes, and join conditions, RecPart always found partitions for which both total input duplication and max worker load were within 10% of the corresponding lower bounds, beating all competitors (even those with significantly higher optimization cost) by a wide margin. Figure 4 shows this for a variety of problems (notice the log scale). The definition of the axes, the algorithms, and the detailed experiments are presented in Section 2, Section 3, and Section 6, respectively.

(4) We prove new Lemmas 2 and 3 that characterize when grid partitioning will be effective for distributed band-joins. Additional material can be found in the extended version of this article [24].

2 PROBLEM DEFINITION

Without loss of generality, let \(S\) and \(T\) be two relations with the same schema \((A_1, A_2, \ldots, A_d)\). Given a band width \(\epsilon_i \geq 0\) for each attribute \(A_i\), the band-join of \(S\) and \(T\) is defined as \(S \Join_B T = \{(s, t) : s \in S \land t \in T \land \forall 1 \leq i \leq d \{ |s.A_i - t.A_i| \leq \epsilon_i \} \). We call \(d\) the dimensionality of the join and \(A_i\) the \(i\)-th dimension. We refer to the \(d\)-dimensional hyper-rectangle centered around a tuple \(a\) with side-length \(2\epsilon_i\) in dimension \(i\), formally \(\{(x_1, \ldots, x_d) : \forall 1 \leq i \leq d \ a.A_i - \epsilon_i \leq x_i \leq a.A_i + \epsilon_i\}\), as the \(\epsilon\)-range around \(a\) (depicted as an orange box in Figure 3).

We call \(d\) the dimensionality of the join and \(A_i\) the \(i\)-th dimension. We refer to the \(d\)-dimensional hyper-rectangle centered around a tuple \(a\) with side-length \(2\epsilon_i\) in dimension \(i\), formally \(\{(x_1, \ldots, x_d) : \forall 1 \leq i \leq d \ a.A_i - \epsilon_i \leq x_i \leq a.A_i + \epsilon_i\}\), as the \(\epsilon\)-range around \(a\) (depicted as an orange box in Figure 3). Note that \((s, t)\) is in the output iff \(s\) falls into the \(\epsilon\)-range around \(t\) (and vice versa). It is straightforward to generalize all results in this paper to asymmetric band conditions \((a.A_i - \epsilon_i \leq x_i \leq a.A_i + \epsilon_i)\) and to relations with attributes that do not appear in the join condition.

**Definition 1 (Join Partitioning).** Given input relations \(S\) and \(T\) with \(Q = S \Join_B T\) and \(w\) worker machines. A join partitioning is an assignment \(h : (S \cup T) \rightarrow 2^{[1, \ldots, w]} \setminus \emptyset\) of each input tuple to one or more workers so that each join result \(q \in Q\) can be recovered by exactly one local join.

A local join is the band-join executed by a worker on the input subset it receives. The definition ensures that each output tuple is produced by exactly one worker, which avoids an expensive post-processing phase for duplicate elimination and is in line with previous work [1, 19, 29, 39]. Given \(S\), \(T\), and a band-join condition, our goal is to minimize the time to compute \(S \Join_B T\). This time is the sum of optimization time (i.e., the time to find a join partitioning) and join time (i.e., the time to compute the result based on this partitioning), as illustrated in Figure 5.

We follow common practice and define the load \(L_i\) on a worker \(w_i\) as the weighted sum \(L_i = \beta_3 I_i + \beta_2 O_i\), \(0 \leq \beta_2, 0 \leq \beta_3\) of input \(I_i\) and output \(O_i\) assigned to it [1, 26, 29, 39]. Max worker load \(L_m = \max_i L_i\) is the maximum load assigned to any worker. In addition, we also evaluate a partitioning based on its total amount of input \(I\). It accounts for given inputs \(S\) and \(T\) and all duplicates created by the partitioning, i.e., \(I = \sum_{x \in S \cup T} |h(x)|\). Recall from **Definition 1** that \(h\) assigns input tuples to a subset of the workers, i.e., \(|h(x)|\) is the number of workers that receive tuple \(x\).

**Lemma 1 (Lower Bounds).** \(|S| + |T|\) is a lower bound for total input \(I\). And \(L_0 = (\beta_2(|S| + |T|)) + \beta_3 |S \Join_B T| / w\) is a lower bound for max worker load \(L_m\).
The lower bound for total input $I$ follows from Definition 1, because each input tuple has to be examined by at least one worker. For max worker load, note that any partitioning has to distribute a total input of at least $|S| + |T|$ and a total output of $|S \times T|$ over the $w$ workers, for a total load of at least $\beta_2(|S| + |T|) + \beta_3|S \times T|$.

**System Model and Measures of Success.** We consider the standard Hadoop MapReduce and Spark environment where inputs $S$ and $T$ are stored in files. These files may have been chunked up before the computation starts, with chunks distributed over the workers. Or they may reside in a separate cloud storage service such as Amazon’s S3. The files are not pre-partitioned on the join attributes and the workers do not have advance knowledge which chunk contains which join-attribute values. Hence any desired join partitioning requires that—in MapReduce terminology—(1) the entire input is read by map tasks and (2) a full shuffle is performed to group the data according to the partitioning and have each group be processed by a reduce task.

In this setting, the shuffle time is determined by total input $I$, and the duration of the reduce phase (each worker performing local joins) is determined by max worker load. Hence we are interested in evaluating how close a partitioning comes to the lower bounds for total input and max worker load (Lemma 1). We use $\frac{I - (|S| + |T|)}{|S| + |T|}$ and $\frac{I_m - L_0}{L_0}$, respectively, which measure by how much a value exceeds the lower bound, relative to the lower bound. For instance, for $I_m = 11$ and $L_0 = 10$ we obtain 0.1, meaning that the max worker load of the partitioning is 10% higher than the lower bound.

For systems with very fast networks, an emerging trend [2, 34], data transfer time is negligible compared to local join time, therefore the goal is to minimize max worker load, i.e., the success measure is $\frac{I_m - L_0}{L_0}$. In applications where the input is already pre-partitioned on the join attributes (e.g., for dimension-dimension array joins [11]) the optimization goal concentrates on reducing data movement [45]. There our approach can be used to find the best pre-partitioning, i.e., to chunk up the array on the dimensions.

In addition to comparing to the lower bounds, we also measure end-to-end running time for a MapReduce/Spark implementation of the band-join in the cloud. For join-time estimation we rely on the model by Li et al. [25], which was shown to be sufficiently accurate to optimize running time of various algorithms, including equi-joins. Similar to the equi-join model, our band-join model $M$ takes as input triple $(I, I_m, O_m)$ and estimates join time as a piecewise linear model $M(I, I_m, O_m) = \beta_0 + \beta_1 I + \beta_2 I_m + \beta_3 O_m$. The $\beta$-coefficients are determined using linear regression on a small benchmark of training queries and inputs.

### Figure 6: Illustration of partitioning methods for band-joins in $d$-dimensional space for $d = 1, 2, 3$; the $A_j$ are the join attributes. Grid-$e$ and RecPart partition the $d$-dimensional join-attribute space, while CSIO and 1-Bucket create partitions by finding a cover of the 2-dimensional join matrix $S \times T$, whose dimensions are independent of the dimensionality of the join condition. Bar height for RecPart and $d = 1$ indicates recursive partition order.

<table>
<thead>
<tr>
<th>Join-Matrix Covering</th>
<th>Attribute-Space Partitioning</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Bucket [29]</td>
<td>CSIO [39]</td>
<td>1</td>
</tr>
<tr>
<td>Grid-$e$ [9, 38]</td>
<td>RecPart</td>
<td>2</td>
</tr>
<tr>
<td>A1</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>A3</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

### 3 RELATED WORK
#### 3.1 Direct Competitors
Direct competitors are approaches that (1) support distributed band-joins and (2) optimize for load balance, i.e., max worker load or a similar measure. We classify them into join-matrix covering vs attribute-space partitioning.

**Join-matrix covering.** These approaches model distributed join computation as a covering problem for the join matrix $J = S \times T$, whose rows correspond to $S$-tuples and columns to $T$-tuples. A cell $J(s, t)$ is “relevant” iff $(s, t)$ satisfies the join condition. Any theta-join, including band-joins, can be represented as the corresponding set of relevant cells in $J$. A join partitioning can then be obtained by covering all relevant cells with non-overlapping regions. Since the exact set of relevant cells is not known a priori (it corresponds to the to-be-computed output), the algorithm covers a larger region of the matrix that is guaranteed to contain all relevant cells. For instance, for inequality predicates, M-Bucket-I [29] partitions both inputs on approximate quantiles in one dimension and then covers with $w$ rectangles all regions corresponding to combinations of inter-quantile ranges from $S$ and $T$ that could potentially contain relevant cells. IEJoin [19] directly uses the same quantile-based range partitioning, but without attempting to find a $w$-rectangle cover. Its main contribution is a clever in-memory algorithm for queries with two join predicates. Optimizing local processing is orthogonal to our focus on how to assign input
tuples to multiple workers. In fact, one can use an adaptation of their idea for local band-join computation on each worker.

To support any theta-join, 1-Bucket [29] covers the entire join matrix with a grid of $r$ rows and $c$ columns. This is illustrated for $r = 3$ and $c = 4$ in Figure 6. Each $S$-tuple is randomly assigned to one of the $r$ rows (which implies that it is sent to all $c$ partitions in this row); this process is analogous for $T$-tuples, which are assigned to random columns. While randomization achieves near-perfect load balance, input is duplicated approximately $\sqrt{w}$ times.

Zhang et al. [44] extend 1-Bucket to joins between many relations. Koumarelas et al. [22] explore re-ordering of join-matrix rows and columns to improve the running time of M-Bucket-I [29]. However, like M-Bucket-I, their technique does not take output distribution into account. This was shown to lead to poor partitionings by Vitorovic et al. [39] whose method CSIO represents the state of the art for distributed theta-joins. It relies on a carefully tuned optimization pipeline that first range-partitions $S$ and $T$ using approximate quantiles, then coarsens those partitions, and finally finds the optimal (in terms of max worker load) rectangle covering of the coarsened matrix. The resulting partitioning was shown to be superior—including for band-joins—to direct quantile-based partitioning, which is used by IEJoin.

Figure 6 illustrates CSIO for a covering with four rectangles for 1, 2 and 3 dimensions. The darker diagonal “bands” show relevant matrix cells, i.e., cells that have to be covered. Notice how join dimensionality affects relevant-cell locations, but does not affect the dimensionality of the join matrix: for a join between two input relations $S$ and $T$, the join matrix is always two-dimensional, with one dimension per input relation.

CSIO suffers from high optimization cost to find the covering rectangles, which uses a tiling algorithm of complexity $O(n^3 \log n)$ for $n$ input tuples. Optimization cost can be reduced by coarsening the statistics used. Further reduction in optimization cost is achieved for monotonic join matrices, a property that holds for 1-dimensional band-joins but not for multidimensional ones. As our experiments will show, the high optimization cost hampers the approach for multidimensional band-joins.

**Attribute-space partitioning.** Instead of using the 2-dimensional $S \times T$ join matrix, attribute-space partitioning works in the $d$-dimensional space $A_1 \times A_2 \times \cdots \times A_d$ defined by the domains of the join attributes. Grid partitioning of the attribute space was explored in the early days of parallel databases, yet only for one-dimensional conditions.

Soloviev [38] proposes the **truncating hash algorithm** and shows that it improves over a parallel implementation of the hybrid partitioned band-join algorithm by DeWitt et al. [9]. The method generalizes to more dimensions as illustrated in the Grid-$\varepsilon$ column in Figure 6. Grid cells define partitions and are assigned to the workers.

By default, Grid-$\varepsilon$ sets grid size for attribute $A_i$ to the band width $\varepsilon_i$ in that dimension. This results in near-zero optimization cost, but may create a poor load balance (for skewed input) and high input duplication (when a partition boundary cuts through a dense region). A coarser grid reduces input duplication, but the larger partitions make load balancing more challenging. Our approach RecPart, which also applies attribute-space partitioning, mitigates the problem by considering recursive partitionings that avoid cutting through dense regions.

### 3.2 Other Related Work

**Similarity joins** are related to band-joins, but neither generalizes the other: the similarity joins closest to band-joins define a pair $(s \in S, t \in T)$ as similar if $\text{sim}(s, t) > \theta$, for some similarity function $\text{sim}$ and threshold $\theta$. This includes 1D band-joins as a special case, but does not support band-joins in multiple dimensions. (A band-join in $d$ dimensions has $2d$ threshold parameters for lower and upper limits in each dimension.) A recent survey [13] compares 10 distributed set-similarity join algorithms. The main focus of previous work on similarity joins is on addressing the specific challenges posed by working in a general metric space where vector-space operations such as addition and scalar multiplication (which band-joins can exploit) are not available. A particular focus is on (1) identifying fast filters that prune away a large fraction of candidate pairs without computing their similarity and (2) selecting pivot elements or anchor points to form partitions, e.g., via sampling [35].

Duggan et al [11] study skew-aware optimization for **distributed array equi-joins** (not band-joins). The work by Zhao et al [45] is closest to ours, because they introduce array similarity joins that can encode multi-dimensional band-join conditions. However, it is not a direct competitor for RecPart, because it considers a different optimization problem: the array is assumed to be already grid-partitioned into chunks on the join attributes and the main challenge is to colocate with minimal network cost those partitions that need to be joined. Our approach is orthogonal for two reasons: First, we do not make any assumptions about existing pre-partitioning on the join attributes and hence the join requires a full data shuffle. Second, we show that for band-joins, grid partitioning is inferior to RecPart’s recursive partitioning. Hence RecPart provides new insights for choosing better array partitions when the array DBMS anticipates band-join queries.

**Attribute-space partitioning** is explored in other contexts for optimization goals that are very different from distributed band-join optimization. For array tiling, the goal is to minimize page accesses of range queries [14]. Here, like
for histogram construction \[32, 33\], the band-join’s data duplication across a split boundary is not taken into account. Histogram techniques optimize for a different goal: maximizing the information captured with a given number of partitions. Only the equi-weight histograms by Vitorovic et al. \[39\] take input duplication into account. We include their approach CS0 in our comparison.

For equi-joins, several algorithms address skew by partitioning heavy hitters \[1, 4, 10, 27, 30, 31, 41, 42\]. Other than the high-level idea of splitting up large partitions to improve load balance, the concrete approaches do not carry over to band-joins: They rely on the property that tuples within band width of a split boundary must be duplicated. However, our decision to use load variance for measuring load balance was inspired by the state-of-the-art equi-join algorithm of Li et al. \[26\]. Earlier work relied on hash partitioning \[8, 21\] and focused on assigning partitions to processors \[7, 12, 17, 18, 20, 34, 40\]. Empirical studies of parallel and distributed equi-joins include \[3, 5, 8, 36, 37\].

### 4 RECURSIVE PARTITIONING

We introduce RecPart and analyze its complexity.

#### 4.1 Main Structure of the Algorithm

RecPart (Algorithm 1) is inspired by decision trees \[16\] and recursively partitions the \(d\)-dimensional space spanned by all join attributes. To adapt this high-level idea to running-time optimization for band-joins, we (1) identify a new splitting measure that determines the selection of split boundaries, (2) propose a new stopping condition for the splitting process, and (3) propose an ordering to determine which tree leaf to consider next for splitting. The splitting process is illustrated in Figure 7.

![Figure 7: Recursive partitioning for a 2D band-join on attributes \(A_1\) and \(A_2\). In the split tree, a path from the root to a leaf defines a rectangular partition in \(A_1 \times A_2\) as the conjunction of all predicates along the path. (By convention the left child is the branch satisfying the split predicate.) In small partitions such as 1.2, RecPart applies 1-Bucket. Those partitions are terminal leaves in the split tree and only change their “internal” partitioning.](image)

**Algorithm 1: RecPart**

**Data:** \(S, T\), band-join condition, sample size \(k\)

**Result:** Hierarchical partitioning \(P^*\) of \(A_1 \times \cdots \times A_d\)

1. Draw random input sample of size \(k/2\) from \(S\) and \(T\)
2. Draw random output sample \[39\] of size up to \(k/2\)
3. Initialize \(P\) with root partition \(P_r = A_1 \times \cdots \times A_d\)
4. \(P_r.\{\text{bestSplit, topScore}\} = \text{best_split}(P_r)\)
5. repeat
   6. Let \(p \in P\) be the leaf node with the highest topScore
   7. Apply \(p.\text{bestSplit}\)
   8. foreach newly created (for regular leaf split) or updated (for small leaf split) leaf node \(p’\) do
      9. \(p’.\{\text{bestSplit, topScore}\} = \text{best_split}(p’)\)
   until termination condition
11. Return best partitioning \(P^*\) found

**Algorithm 2: best_split**

**Data:** Partition \(p\), input and output sample tuples in \(p\)

1. number of row sub-partitions \(r\) and column sub-partitions \(c\) (\(r = c = 1\) for regular partitions)

**Result:** Split predicate \(\text{bestSplit}\) and its score \(\text{topScore}\)

1. Initialize \(\text{topScore} = 0\) and \(\text{bestSplit} = \text{NULL}\)
2. if \(p\) is a regular partition
   3. // Find best decision-tree style split
      4. foreach regular dimension \(A_i\) do
         5. Let \(x_i\) be the split predicate on dimension \(A_i\) that has the highest ratio \(\sigma_i = \Delta \text{Var}(x_i)/\Delta \text{Dup}(x_i)\) among all possible splits in dimension \(A_i\)
         6. if \(\sigma_i > \text{topScore}\) then
            7. Set \(\text{topScore} = \sigma_i\) and set \(\text{bestSplit} = x_i\)
      8. else
         9. Set \(\text{topScore} = \sigma_c\) and set \(\text{bestSplit} = \text{column}\)
   10. Return \(\text{bestSplit}, \text{topScore}\)
on this leaf to find the best possible split and its score. Assuming the leaf is a "regular" partition (we discuss small partitions below), best_split sorts the input sample on \( A_1 \) and tries all middle-points between consecutive \( A_1 \)-values as possible split boundaries. Then it does the same for \( A_2 \). The winning split boundary is the one with the highest ratio between load-variance reduction and input duplication increase (see details in Section 4.2).

Assume the best split is \( A_2 < a \). The first execution of the repeat-loop then applies this split, creating new leaves "1" and "2" and finding the best split for each of them. Which leaf should be split next? **RecPart manages all split-tree leaves in a priority queue based on their topScore value.** Assuming leaf "1" has the higher score, the next repeat-loop iteration will split it, creating new leaves "1.1" and "1.2". This process continues until the appropriate termination condition is reached (discussed below). As the split tree is grown, the algorithm also keeps track of the best partitioning found so far.

### 4.2 Algorithm Details

**Small partitions.** When a partition becomes "small" relative to bandwidth in a dimension, then no further recursive splitting in that dimension is allowed. When the partition is small in all dimensions, then it switches into a different partitioning mode inspired by 1-Bucket [29]. This is motivated by the observation that when the length of a partition approaches bandwidth in each dimension, then all \( S \) and \( T \)-tuples in that partition join with each other. And for Cartesian products, 1-Bucket was shown to be near-optimal.

We define a partition as "small" as soon as its size is below twice the band width in all dimensions. In Figure 7 step (4), leaf "1.2" is small and hence when it is picked in the repeat-loop, applying the best split leaves the split tree unchanged, but instead increases the number of column partitions \( c \) to 2. Afterward, the topScore value of leaf "1.2" may have decreased and leaf "2" is split next, using a regular recursive split. This may be followed by more "internal" splits of leaf "1.2" in later iterations as shown in Figure 7, steps (6) and (7).

It is easy to show that having some leaves in "regular" and others in "small" split mode does not affect correctness. Intuitively, this is guaranteed because duplication only needs to be considered inside the region that is further partitioned, i.e., it does not "bleed" beyond partition boundaries.

**Split scoring.** In split score \( \Delta \text{Var}(x_i)/\Delta \text{Dup}(x_i) \) for a regular dimension (Algorithm 2), \( \Delta \text{Var}(x_i) \) is defined as follows: Let \( L \) denote the set of split-tree leaves. Each (sub-)partition in \( a \) leaf corresponds to a region \( p \) in \( A_1 \times \cdots \times A_d \), for which we estimate input \( I_p \) and output \( O_p \) from the random samples drawn by Algorithm 1. The load induced by \( p \) is \( I_p = \beta_2 I_p + \beta_3 O_p \). Load variance is computed as follows: Assign each leaf in \( L \) to a randomly selected worker. Then per-worker load is a random variable \( P \) (we slightly abuse notation to avoid notational clutter) whose variance can be shown to be \( \Delta \text{Var}(P) = \sum_{p \in L} \frac{I_p^2}{w_p} \). We analogously obtain \( \Delta \text{Var}(P') \) for a partitioning \( P' \) that results from splitting some leaf \( p' \in L \) into sub-partitions \( p_1 \) and \( p_2 \) using predicate \( x_j \). Then \( \Delta \text{Var}(x_j) = \Delta \text{Var}(P') - \Delta \text{Var}(P) \). \( \Delta \text{Var}(P') \) can be computed from \( \Delta \text{Var}(P) \) in constant time by subtracting \( \sum_{p \in L} \frac{I_p^2}{w_p} \) and adding \( \sum_{p \in L} \frac{I_{p'}^2}{w_{p'}} \).

The additional duplication caused by a split is obtained by estimating the number of \( T \)-tuples within band width of the new split boundary using the input sample. When multiple split predicates cause no input duplication, then the best split is the one with the greatest variance reduction among them. The calculation of load variance and input duplication for "small" leaves is analogous.

The split score reflects our goal of reducing max worker load with minimal input duplication. For the former, load-variance reduction could be replaced by other measures, but precise estimation of input and output on the most loaded worker is difficult due to dynamic load balancing applied by schedulers at runtime. We therefore selected load variance as a scheduler-independent proxy.

**Termination condition and winning partitioning.** We propose a theoretical and an applied termination condition for the repeat-loop in Algorithm 1. For the theoretical approach, the winning partitioning is the one with the lowest overhead over the lower bound in terms of both max worker load and input duplication, i.e., the one with the minimal value of \( \min \left\{ \frac{1 - (\frac{|S| + |T|)}{|S| + |T|}}{\beta_2 \frac{L_m - L_a}{L_a}}, \frac{L_m - L_a}{L_a} \right\} \). It is easy to show that each iteration of the repeat-loop monotonically increases input \( I \), because each new split boundary (regular leaf) and more fine-grained sub-partitioning (small leaf) can only increase the number of input duplicates. At the same time, the loop iteration may or may not decrease \( L_m - L_a \). Hence repeat-loop iterations can be terminated as soon as \( \frac{1 - (\frac{|S| + |T|)}{|S| + |T|}}{\beta_2 \frac{L_m - L_a}{L_a}} \) exceeds the smallest value of \( \frac{L_m - L_a}{L_a} \) encountered so far.

The theoretical approach only needs input and output samples, as well as an estimate of the relative impact of an input tuple versus an output tuple on local join computation time. Input sampling is straightforward; for output sampling we use the method from [39]. If the output is large, it efficiently produces a large sample. If the output is small, then the output has negligible impact on join computation cost. The experiments show that we get good results when limiting sample size based on memory size and sampling cost to at most 5% of join time.

For estimating load impact, we run band-joins with different input and output sizes \( I \) and \( O \) on an individual worker and use linear regression to determine \( \beta_2 \) and \( \beta_3 \) in load function \( \beta_2 I + \beta_3 O \). In our Amazon cloud cluster, \( \beta_2/\beta_3 \approx 4 \). Note that \( \beta_2 \) and \( \beta_3 \) tend to increase with input and output. For the
lower bound, we use the smallest values, i.e., those obtained for scenarios where a node receives about $1/w$ of the input (recall that $w$ is the number of workers). This establishes a lower value for the lower bound, i.e., it is more challenging for RecPart to be close to it.

For the applied approach, we use the cost model as discussed in the end of Section 2. The winning partitioning is the one with the lowest running time predicted by the cost model. Repeat-loop iterations terminate when estimated join time bottoms out. We detect this based on a window of the join times over the last $w$ repeat-loop iterations: loop execution terminates when improvement is below 1% (or join times over the last $w$ iterations). (We chose $w$ as window size because it would take at least one extra split per worker to break up each worker’s load. This dovetails with the prioritization of leaves: The most promising leaves in terms of splitting up load with low input duplication overhead are greedily selected.)

**Extension: symmetric partitioning.** Like classic grid partitioning, RecPart as discussed so far treats inputs $S$ and $T$ differently: at an inner node in the split tree, $S$ is partitioned (without duplication), while $T$-tuples near the split boundary are duplicated. For regions where $S$ is sparse and $T$ is dense, we want to reverse these roles. Consider Figure 2c, where $y_1$ and $y_2$ enabled a zero-duplication partitioning with perfect load balance. What if the input distribution was reversed in another region of the join-attribute space, e.g., $S’ = \{21, 25, 26, 30\}$ and $T’ = \{21, 22, 23, 25, 26, 28, 29, 30\}$? Then no split in range 21 to 30 could avoid duplication of $T’$-tuples because for band width 1 at least one of the $T’$-values would be within 1 of the split point. In that scenario we want to reverse the roles of $S’$ and $T’$, i.e., perform the partitioning on $T’$ and the partition/duplication on $S’$.

For grid partitioning, it is not clear how to reverse the roles of $S$ and $T$ in some of the grid cells. For RecPart, this turns out to be easy. When exploring split options in a regular leaf (1-Bucket in small partitions already treats both inputs symmetrically), Algorithm 2 computes the duplication for both cases: partition $S$ and partition/duplicate $T$ as well as the other way round. We call the former a $T$-split and the latter an $S$-split. The split type information is added to the corresponding node in the split tree.

Algorithm 3 is used to determine which tuples to assign to the sub-trees. (The version for $T$-tuples is shown; the one for $S$-tuples is analogous.) It is easy to show that for each result $(s, t) \in S \bowtie B T$, exactly one leaf in the split tree receives both $s$ and $t$.

### 4.3 Algorithm Analysis

RecPart has low complexity, resulting in low optimization time. Let $\lambda$ denote the number of repeat-loop executions in

<table>
<thead>
<tr>
<th>Algorithm 3: assign_input</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data:</strong> Input tuple $t \in T$; node $p$ in split tree</td>
</tr>
<tr>
<td><strong>Result:</strong> Set of leaves to which $t$ is copied</td>
</tr>
<tr>
<td>1. <strong>if</strong> $p$ is a leaf <strong>then</strong></td>
</tr>
<tr>
<td>2. Return (1-Bucket $(t, p)$)</td>
</tr>
<tr>
<td>3. <strong>else</strong></td>
</tr>
<tr>
<td>4. <strong>if</strong> $p$ is a $T$-split node <strong>then</strong></td>
</tr>
<tr>
<td>5. For each child partition $p'$ of $p$ that intersects with the $\epsilon$-range of $t$ do</td>
</tr>
<tr>
<td>6. assign_input$(t, p')$</td>
</tr>
<tr>
<td>7. <strong>else</strong></td>
</tr>
<tr>
<td>8. Let $p'$ be the child partition of $p$ that contains $t$</td>
</tr>
<tr>
<td>9. assigning_input$(t, p')$</td>
</tr>
</tbody>
</table>

Algorithm 1. Each iteration increases the number of leaves in the split tree by at most one. The algorithm manages all leaves in a priority queue based on the score returned by Algorithm 2. With a priority queue where inserts have constant cost and removal of the top element takes time logarithmic in queue size, an iteration of the repeat-loop takes $O(\log \lambda)$ to remove the top-scoring leaf $p$. If $p$ is regular, then it takes $O(1)$ to create sub-partitions $p_1$ and $p_2$ and to distribute all input and output samples in $p$ over them. (Sample size is bounded by constant $k$, a fraction of machine-memory size.) Checking if $p_1$ and $p_2$ are small partitions takes time $O(d)$. Then best_split is executed, which for a regular leaf requires sorting of the input sample on each dimension and trying all possible split points, each a middle point between two consecutive sample tuples in that dimension. Since sample size is upper-bounded by a constant, the cost is $O(d)$. For a small leaf, the cost is $O(1)$. Finally, inserting $p_1$ and $p_2$ into the priority queue takes $O(1)$. In total, splitting a regular or small leaf has complexity $O(d)$ and $O(1)$, respectively.

After $\lambda$ executions of the repeat-loop in Algorithm 1, the next iteration has complexity $O(\log \lambda + d)$. Hence the total cost of $\lambda$ iterations is $O(\lambda \log \lambda + \lambda d)$. In our experience, the algorithm will terminate after a number of iterations bounded by a small multiple of the number of worker machines. To see why, note that each iteration breaks up a large partition $p$ and replaces it by two (for regular $p$) or more (for small $p$) sub-partitions. The split-scoring metric favors breaking up heavy partitions, therefore load can be balanced across workers fairly evenly as soon as the total number of partitions reaches a small multiple of the number of workers. This in turn implies for Algorithm 1, given samples of fixed size, a total complexity of $O(w \log w + wd)$.

### 5 ANALYTICAL INSIGHTS

We present two surprising results about the ability of grid partitioning to address load imbalances. For join-matrix covering approaches like CS90 that depend on the notion of a
total ordering of the join-attribute space, we explore how to enumerate the multi-dimensional space.

5.1 Properties of Grid Partitioning

Without loss of generality, let S be the input that is partitioned and T be the input that is partitioned/duplicated.

Input duplication. With grid-size in each dimension set to the corresponding band width, the ε-range of a T-tuple intersects with up to 3 grid cells per dimension, for a total replication rate of $O(3^d)$ in d dimensions. Can this exponential dependency on the dimensionality be addressed through coarser partitioning? Unfortunately, for any non-trivial partitioning, asymptotically the replication rate is still at least $O(2^d)$. To construct the worst case, an adversary places all input tuples near the corner of a centrally located grid cell.

Max worker load. We now show an even stronger negative result, indicating that grid-partitioning is inherently limited in its ability to reduce max worker load, no matter the number of workers or the grid size. Consider a partition where one of the grid cells contains half of S and half of T. Does there exist a more fine-grained grid partitioning where none of the grid cells receives more than 10% of S and 10% of T? One may be tempted to answer in the affirmative: just keep decreasing grid size in all dimensions until the target is reached. Unfortunately, this does not hold as we show next.

Lemma 2. If there exists an ε-range in the join-attribute space with n tuples from T, then grid partitioning will create a partition with at least n T-tuples, no matter the grid size.

Proof. For d = 1 consider an interval of size ε1 that contains n T-tuples. If no split point partitions the interval, then the partition containing it has all n T-tuples. Otherwise, i.e., if at least one split point partitions the interval, pick one of the split points inside the interval, say X, and consider the T-tuples copied to the grid cells adjacent to X. Since all T-tuples in an interval of size ε1 are within ε1 of X, both grid cells receive all n T-tuples from the interval. It is straightforward to generalize this analysis to $d > 1$.

In short, even though a more fine-grained partitioning can split S into ever smaller pieces, the same is not possible for T because of the duplication needed to ensure correctness for a band-join. For skewed input, n in Lemma 2 can be $O(|T|)$. Interestingly, we now show that grid partitioning can behave well in terms of load distribution for skewed input, as long as the input is "sufficiently" large.

Lemma 3. Let $c_0 > 0$ be a constant such that $|S \bowtie R, T| \leq c_0(|S| + |T|)$. Let R (R') denote the region of size $\epsilon_1 \times \epsilon_2 \times \cdots \times \epsilon_d$ in the join attribute space containing the most tuples from S (T); and let $x (x')$ and $y (y')$ denote the fraction of tuples from S and T, respectively, it contains. If there exist constants $0 < c_1 \leq c_2$ such that $c_1 \leq x / y \leq c_2$ and $c_1 \leq x' / y' \leq c_2$, then no region of size $\epsilon_1 \times \epsilon_2 \times \cdots \times \epsilon_d$ contains more than $O(\sqrt[1/3]{|S| + 1/|T|})$ input tuples.

Proof. By definition, all S and T tuples in region R join with each other. Together with $|S \bowtie R, T| \leq c_0(|S| + |T|)$ this implies

$$x|S| \cdot y |T| \leq c_0(|S| + |T|) \implies xy \leq c_0(1/|S| + 1/|T|)$$

(1)

From $x/y \leq c_2$ follows $x^2 \leq c_2xy$, then $x^2 \leq c_2c_0(1/|S| + 1/|T|)$ (from (1)) and thus $x \leq \sqrt{c_0c_2\sqrt{1/|S| + 1/|T|}} = O(\sqrt[1/3]{1/|S| + 1/|T|})$. We show analogously for region $R'$ that $y' \leq \sqrt{c_0/c_1}(\sqrt[1/3]{|S| + 1/|T|}) = O(\sqrt[1/3]{1/|S| + 1/|T|})$.

Since $R$ is the region of size $\epsilon_1 \times \epsilon_2 \times \cdots \times \epsilon_d$ with most S-tuples and $R'$ the region with most T-tuples, no region of size $\epsilon_1 \times \epsilon_2 \times \cdots \times \epsilon_d$ can contain more than $x$ fraction of S-tuples and $y'$ fraction of T-tuples.

Lemma 3 is surprising. It states that for larger inputs S and T, the fraction of S and T in any partition of Grid-ε (recall that its partitions are of size $\epsilon_1 \times \epsilon_2 \times \cdots \times \epsilon_d$) is upper-bounded by a function that decreases proportionally with $\sqrt[1/3]{|S|}$ and $\sqrt[1/3]{|T|}$. For instance, when S and T double in size, then the upper bound on the input fraction in any partition decreases by a factor of $\sqrt[2]{2} \approx 1.4$.

Clearly, this does not hold for all band joins. The proof of Lemma 3 required (1) that the region with most S-tuples contain a sufficiently large fraction of T, and vice versa; and (2) that output size is bounded by $c_0$ times input size. The former is satisfied when S and T have a similar distribution in join-attribute space, e.g., a self-join. For the latter, we are aware of two scenarios. First, the user may actively try to avoid huge outputs by setting a smaller band width. Second, for any output-cost-dominated theta-join, 1-Bucket was shown to be near-optimal [29]. Hence specialized band-join solutions such as RecPart, Grid-ε, and CSIO would only be considered when output is "sufficiently" small.

5.2 Ranges in Multidimensional Space

CSIO starts with a range partitioning of the join-attribute space based on approximate quantiles. For quantiles to be
well-defined, a total order must be established on the multi-dimensional space—this was left unspecified for $C_{90}$. The ordering can significantly impact performance as illustrated in Figure 8 for a 2D band-join. In row-major order, ranges correspond to long horizontal stripes. Alternatively, each range may correspond to a “block,” creating more square-shaped regions. This choice affects the candidate regions in the join matrix that need to be covered. Assume each horizontal stripe in Figure 8a is at least $\varepsilon_1$ high. Then an $S$-tuple in stripe $i$ can only join with $T$-tuples in the three stripes $i-1$, $i$, and $i+1$. This creates the compact candidate-region diagonal of width 3 in the join matrix. For the block partitioning in Figure 8b, S-tuples in a block may join with T-tuples in up to nine neighboring blocks. This creates the wider band in the join matrix. For $C_{90}$, a wider diagonal and denser matrix cause additional input duplication.

In general, the number of candidate cells in the join matrix is minimized by row-major ordering, if the distance between the hyperplanes in the most significant dimension is greater than or equal to the band width in that dimension. This was the case in our experiments and therefore row-major ordering was selected for $C_{90}$.

6 EXPERIMENTS

We compare the total running time (optimization plus join time) of RecPart to the state of the art (GRID-$\varepsilon$, $C_{90}$, 1-Bucket). Note that GRID-$\varepsilon$ is not defined for band width zero. Reported times are measured on a real cloud, unless stated otherwise. In large tables, we mark cells with the main results in blue; red color highlights a weak spot, e.g., excessive optimization time or input duplication.

6.1 Experimental Setup

Environments. Both MapReduce [6] and Spark [43] are well-suited for band-join implementation. Spark’s ability to keep large data in memory makes little difference for the map-shuffle-reduce pipeline of a band-join, therefore we use MapReduce, where it is easier to control low-level behavior such as custom data partitioning. All experiments were conducted on Amazon’s Elastic MapReduce (EMR) cloud, using 30 m3. xlarge machines (15GB RAM, 40GB disk, high network performance) by default. All clusters run Hadoop 2.8.4 with the YARN scheduler in default configuration.

Data. For synthetic data, we use a Pareto distribution where join-attribute value $x$ is drawn from domain $[1, 0, \infty)$ of real numbers and follows PDF $z/x^{x+1}$ (greater $z$ creates more skew). This models the famous power-law distribution observed in many real-world contexts, including the 80-20 rule for $z = \log z$, $5 \approx 1.16$. We explore $z$ in the range $[0.5, 2.0]$, which covers power-law distributions observed in real data. pareto-z denotes a pair of tables, each with 200 million tuples, with Pareto-distributed join attributes for skew $z$. High-frequency values in $S$ are also high-frequency values in $T$. rv-pareto-$z$ is the same as pareto-$z$, but high-frequency values in $S$ have low frequency in $T$, and vice versa. Specifically, $T$ follows a Pareto distribution from $10^6$ down to $-\infty$ ($T$ is skewed toward larger values. We generate $T$ by drawing numbers from $[1.0, \infty)$ following Pareto distribution and then converting each number $y$ to $10^6 - y$.)

cloud is a dataset containing 382 million cloud reports [15], each reporting time, latitude, longitude, and 25 weather attributes. ebird is a real dataset containing 508 million bird sightings, each with attributes describing time, latitude, longitude, species observed, and 1655 features of the observation site [28].

For each input, we explore different band widths as summarized in Table 1. Output sizes below 0.5 million are reported as 0. For the real data, the three join attributes are time ([days] since January 1st, 1970), latitude ([degrees] between $-90$ and 90) and longitude ([degrees] between $-180$ and 180).

Local join algorithm. After partitions are assigned to workers, each worker needs to locally perform a band-join on its partition(s). Many algorithms could be used, ranging from nested-loop to adaptations of IEJoin’s sorted arrays and bit-arrays. Since we focus on the partitioning aspect, the choice of local implementation is orthogonal, because it only affects the relative importance of optimizing for input duplication vs optimizing for max worker load. We observed that RecPart wins no matter what this ratio is set to, therefore

<table>
<thead>
<tr>
<th>Data Set</th>
<th>$d$</th>
<th>Band width</th>
<th>Input Size</th>
<th>Output Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>pareto-1.5</td>
<td>1</td>
<td>0</td>
<td>400</td>
<td>2430</td>
</tr>
<tr>
<td>pareto-1.5</td>
<td>1</td>
<td>$10^{-5}$</td>
<td>400</td>
<td>4580</td>
</tr>
<tr>
<td>pareto-1.5</td>
<td>1</td>
<td>$2 \cdot 10^{-5}$</td>
<td>400</td>
<td>9120</td>
</tr>
<tr>
<td>pareto-1.5</td>
<td>1</td>
<td>$3 \cdot 10^{-5}$</td>
<td>400</td>
<td>11280</td>
</tr>
<tr>
<td>pareto-1.5</td>
<td>3</td>
<td>(0, 0, 0)</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>pareto-1.5</td>
<td>3</td>
<td>(2, 2, 2)</td>
<td>400</td>
<td>1120</td>
</tr>
<tr>
<td>pareto-1.5</td>
<td>3</td>
<td>(4, 4, 4)</td>
<td>400</td>
<td>8740</td>
</tr>
<tr>
<td>pareto-0.5</td>
<td>3</td>
<td>(2, 2, 2)</td>
<td>400</td>
<td>12</td>
</tr>
<tr>
<td>pareto-1.0</td>
<td>3</td>
<td>(2, 2, 2)</td>
<td>400</td>
<td>420</td>
</tr>
<tr>
<td>pareto-2.0</td>
<td>3</td>
<td>(2, 2, 2)</td>
<td>400</td>
<td>3200</td>
</tr>
<tr>
<td>pareto-1.5</td>
<td>8</td>
<td>(20, \ldots , 20)</td>
<td>100</td>
<td>9</td>
</tr>
<tr>
<td>pareto-1.5</td>
<td>8</td>
<td>(20, \ldots , 20)</td>
<td>200</td>
<td>57</td>
</tr>
<tr>
<td>pareto-1.5</td>
<td>8</td>
<td>(20, \ldots , 20)</td>
<td>400</td>
<td>219</td>
</tr>
<tr>
<td>pareto-1.5</td>
<td>8</td>
<td>(20, \ldots , 20)</td>
<td>800</td>
<td>857</td>
</tr>
<tr>
<td>rv-pareto-1.5</td>
<td>1</td>
<td>2</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>rv-pareto-1.5</td>
<td>1</td>
<td>1000</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>rv-pareto-1.5</td>
<td>3</td>
<td>(1000, 1000, 1000)</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>rv-pareto-1.5</td>
<td>3</td>
<td>(2000, 2000, 2000)</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>ebird and cloud</td>
<td>3</td>
<td>(0, 0, 0)</td>
<td>890</td>
<td>0</td>
</tr>
<tr>
<td>ebird and cloud</td>
<td>3</td>
<td>(1, 1, 1)</td>
<td>890</td>
<td>320</td>
</tr>
<tr>
<td>ebird and cloud</td>
<td>3</td>
<td>(2, 2, 2)</td>
<td>890</td>
<td>2134</td>
</tr>
<tr>
<td>ebird and cloud</td>
<td>3</td>
<td>(4, 4, 4)</td>
<td>890</td>
<td>16998</td>
</tr>
</tbody>
</table>
in our experiments we selected a fairly standard local band-join algorithm based on index-nested-loops. (Let $S_p$ and $T_p$ be the input in partition $p$.) (1) range-partition $T_p$ on $A_1$ into ranges of size $\epsilon_1$. (Here $A_1$ is the most selective dimension.) (2) For each $s \in S_p$, use binary search to find the $T$-range $i$ containing $s$. Then check band condition on $(s, t)$ for all $t$ in ranges $(i - 1), i, \text{and } i + 1$.

Since Grid-$\epsilon$ partitions are of size $\epsilon_1$ in dimension $A_1$, we slightly modify the above algorithm and sort both $S_p$ and $T_p$ on $A_1$. The binary search for $s \in S_p$ then searches for $s.A_1 - \epsilon_1$ (the smallest tuple $t \in T_p$ it could join with) and scans the sorted $T$-array from there until $s.A_1 + \epsilon_1$.

Statistics and running-time model. We sample 100,000 input records and set output-sample size so that total time for statistics gathering does not exceed 5% of the fastest time (optimization plus join time) observed for any method. For output sampling we use the method introduced for CSIO [39]. For the cost model (see Section 2), we determine the model coefficients ($\beta$-values) as discussed in [25] from a benchmark of 100 queries. The benchmark is run offline once to profile the performance aspects of a cluster.

6.2 Impact of Band Width
We explore the impact of band width, which affects output. For comparison with grid partitioning, we turn RecPart’s symmetric partitioning off, i.e., $T$ is always the partitioned/duplicated relation. Since the grid approaches do not apply symmetric partitioning by design, all advantages of RecPart in the experiments are due to the better partition boundaries, not due to symmetric partitioning. (The impact of symmetric partitioning is explored separately later.) To avoid confusion, we refer to RecPart without symmetric partitioning as RecPart-S.

6.2.1 Single Join Attribute. The left block in Table 2a reports running times. RecPart-S wins in all cases, by up to a factor of 2, but the other methods are competitive because the join is in 1D and skew is moderate. For CSIO we tried different parameter settings that control optimization time versus partitioning quality, reporting the best found. The right block in Table 2a reports input plus duplicates ($I$), and the input ($I_m$) and output ($O_m$) on the most loaded worker machine. Recall that profiling revealed $\beta_2/\beta_3 \approx 4$, i.e., each input tuple incurs 4 times the load compared to an output tuple.

RecPart-S and CSIO achieve similar load characteristics, because both intelligently leverage input and output samples to balance load while avoiding input duplication. However, RecPart-S finds a better partitioning (lower max worker load and input duplication) with 10x lower optimization time. The other two methods produce significantly higher input duplication, affecting $I$ and $I_m$. Grid-$\epsilon$ still shows competitive running time because it works with a very fine-grained partitioning, i.e., each worker receives its input already split into many small grid cells. Data in a cell can be joined independent of the other cells, resulting in efficient in-memory processing. As a result, Grid-$\epsilon$ has lower per-tuple processing time than the other methods. (There each worker receives its input in a large “chunk” that needs to be range-partitioned.)

6.2.2 Multiple Join Attributes. Tables 2b and 2c show that the performance gaps widen when joining on 3 attributes: RecPart-S is the clear winner in total running time as well as join time alone. It finds the partitioning with the lowest max worker load, while keeping input duplication below 4%, while the competitors created up to 12x input duplication.

CSIO is severely hampered by the complexity of the optimization step. (Lowering optimization time resulted in higher join time due to worse partitioning.) Grid-$\epsilon$ suffers from $O(3^d)$ input duplication in $d$ dimensions. For 1-Bucket, note that the numbers in Table 2a and Table 2b are virtually identical. This is due to the fact that it covers the entire join matrix $S \times T$, i.e., the matrix cover is not affected by the dimensionality of the join condition.

Grid-$\epsilon$ has by far the highest input duplication, but again recovers some of this cost due to its faster local processing that exploits that each worker’s input arrives already partitioned into small grid cells. This is especially visible when comparing to 1-Bucket for band width $(2, 2, 2)$ in Table 2c.

6.3 Skew Resistance
Table 3 investigates the impact of join-attribute skew, showing that RecPart-S handles it the best, again achieving the lowest max worker load with almost no input duplication. The competitors suffer from high input duplication; CSIO also from high optimization cost. Note that as skew increases, output size increases as well. This is due to the power-law distribution and the correlation of high-frequency join-attribute values in the two inputs. Greater output size implies a denser join matrix for CSIO, increasing its optimization time.

6.4 Scalability
Tables 4a to 4d show that RecPart-S and RecPart have almost perfect scalability and beat all competitors. In Tables 4a and 4b, from row to row, we double both input size and number of workers. In Table 4c, only the input size varies while the number of workers is constant. In Table 4d, we only change the number of workers. The latter two results are for an 8D band-join to explore which techniques can scale beyond dimensionality common today. For cost reasons, we use the running-time model to predict join time in Tables 4c and 4d. For queries on real data, the smaller inputs are random samples from the full data. Note that join output grows super-linearly, therefore perfect scalability cannot be
Table 2: Impact of band width: RecPart-S wins in all cases, and the winning margin gets bigger for band joins with more dimensions. (Blue color highlights the main results; red color highlights a weak spot.)

(a) Pareto-1.5, d = 1, varying band width.

<table>
<thead>
<tr>
<th>Band width</th>
<th>Runtime (optimization time+join time) in [sec]</th>
<th>Relative time over RecPart-S</th>
<th>I/O sizes in [millions]: I, I_m, O_m</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RecPart-S</td>
<td>CSIO</td>
<td>1-Bucket</td>
</tr>
<tr>
<td>0</td>
<td>351(3+348)</td>
<td>512(29+483)</td>
<td>762</td>
</tr>
<tr>
<td>10^-5</td>
<td>539(7+532)</td>
<td>684(29+655)</td>
<td>1004</td>
</tr>
<tr>
<td>2^-5</td>
<td>813(3+810)</td>
<td>992(30+962)</td>
<td>1316</td>
</tr>
<tr>
<td>3^-5</td>
<td>878(3+875)</td>
<td>1170(30+1140)</td>
<td>1520</td>
</tr>
</tbody>
</table>

(b) Pareto-1.5, d = 3, varying band width.

<table>
<thead>
<tr>
<th>Band width</th>
<th>Runtime (optimization time+join time) in [sec]</th>
<th>Relative time over RecPart-S</th>
<th>I/O sizes in [millions]: I, I_m, O_m</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RecPart-S</td>
<td>CSIO</td>
<td>1-Bucket</td>
</tr>
<tr>
<td>(0, 0, 0)</td>
<td>230(1+229)</td>
<td>366(46+320)</td>
<td>792</td>
</tr>
<tr>
<td>(2, 2, 2)</td>
<td>344(2+342)</td>
<td>1339(694+645)</td>
<td>1149</td>
</tr>
<tr>
<td>(4, 4, 4)</td>
<td>860(2+858)</td>
<td>2557(1345+1212)</td>
<td>1772</td>
</tr>
</tbody>
</table>

(c) Join of ebird with cloud, d = 3, varying band width.

<table>
<thead>
<tr>
<th>Band width</th>
<th>Runtime (optimization time+join time) in [sec]</th>
<th>Relative time over RecPart-S</th>
<th>I/O sizes in [millions]: I, I_m, O_m</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RecPart-S</td>
<td>CSIO</td>
<td>1-Bucket</td>
</tr>
<tr>
<td>(0, 0, 0)</td>
<td>248(3+245)</td>
<td>346(38+308)</td>
<td>1418</td>
</tr>
<tr>
<td>(1, 1, 1)</td>
<td>352(3+329)</td>
<td>1945(968+977)</td>
<td>1552</td>
</tr>
<tr>
<td>(2, 2, 2)</td>
<td>423(3+420)</td>
<td>2615(1553+1062)</td>
<td>1573</td>
</tr>
</tbody>
</table>

Table 3: Skew resistance: RecPart-S is fastest and has much less data duplication than other methods. (Pareto-z, d = 3, band width (2, 2, 2), and increasing skew z = 0.5, . . . , 2.)

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Runtime (optimization time+join time) in [sec]</th>
<th>Relative time over RecPart-S</th>
<th>I/O sizes in [millions]: I, I_m, O_m</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RecPart-S</td>
<td>CSIO</td>
<td>1-Bucket</td>
</tr>
<tr>
<td>pareto-0.5</td>
<td>230(3+227)</td>
<td>609(263+346)</td>
<td>1137</td>
</tr>
<tr>
<td>pareto-1.0</td>
<td>290(3+287)</td>
<td>1064(525+539)</td>
<td>1235</td>
</tr>
<tr>
<td>pareto-1.5</td>
<td>344(2+342)</td>
<td>1339(694+645)</td>
<td>1149</td>
</tr>
<tr>
<td>pareto-2.0</td>
<td>485(2+483)</td>
<td>1811(1008+811)</td>
<td>1369</td>
</tr>
</tbody>
</table>

6.5 Optimizing Grid Size

Table 5 shows that grid granularity has a significant impact on join time, here model estimated, of Grid-ε. With the default grid size (2, 2, 2), join time is 9x higher compared to grid size (32, 32, 32), caused by input duplication. Our extension Grid+ automatically explores different grid sizes, using the same running-time model M as RecPart and CSIO to find the best setting. Starting with grid size ε_l in dimension i for all join attributes A_i, it tries coarsening the grid to size j · ε_l in dimension i, for j = 2, 3, . . . . For each resulting grid partitioning G, we execute M(G) to let model M predict the running time, until a local minimum is found.

Automatic grid tuning works well for Pareto-z where both inputs are similarly distributed and band width is small: Lemma 3 applies with large c_1 and small c_2, providing strong upper bounds on the amount of input in any ε-range. This confirms that grid-partitioning can indeed work well for “sufficiently large” input even in 3D space. However, Grid+ fails on the reverse Pareto distribution as Table 6 shows. There S and T have very different density, resulting in small c_1 and large c_2, and therefore much weaker upper bounds on the input per ε-range. The resulting dense regions, as stated by Lemma 2, cause high input duplication and high input I_m assigned to the most loaded worker.

6.6 Comparing to Distributed IEJoin

Table 7 shows representative results for a comparison to the quantile-based partitioning used by IEJoin [19]. We explore a wide range of inter-quantile ranges (sizePerBlock) and report results for those at and near the best setting found. Note that RecPart can use the same local processing algorithm as IEJoin, therefore we are interested in comparing based on the quality of the partitioning, i.e., I, I_m, and O_m. It is clearly visible that RecPart-S finds significantly better partitionings, providing more evidence that simple quantile-based partitioning of the join matrix does not suffice.
6.7 Impact of Local Join Algorithm

Table 8 shows a typical result of the impact of ratio $\beta_2/\beta_1$. A high ratio occurs in systems with fast data transfer and slow local computation. Replacing the local band-join algorithm with a faster one would lower the ratio. While the competitors are not affected by the ratio (they all ignore network shuffle time), it is visible how increasing weight on local

6.8 Impact of Symmetric Partitioning

While RecPart and RecPart-S find similar partitionings on pareto-1.0 and the real data, the advantages of symmetric partitioning are revealed on the reverse Pareto data in Table 9. Here RecPart-S cannot split regions with high density of $T$ without incurring high input duplication. In
6.10 Near-Optimality of Our Approach

Figure 4 in Section 1 summarizes the results from all tables, making it easy to see how RecPart achieves significantly lower max worker load (y-axis) with less input duplication (x-axis). RecPart is always within 10% of the lower bound on both measures, beating the competition by a wide margin.

7 CONCLUSIONS

For distributed band-joins, we showed that recursive partitioning with the appropriate split-scoring measure achieves both low optimization time (a few seconds) and near-optimal join partitioning (within 10% of the respective lower bounds) on input duplication and max worker load. Even if system parameters change, e.g., due to faster networks or CPUs, RecPart’s achievement will still stand, because the issues of low input duplication and low input and output per worker will remain crucial optimization goals.

There are several exciting directions for future work. First, for band-joins between more than two relations, can one do better than reducing the problem to multiple two-way joins? Second, how do we partition for join conditions that contain a mix of various types of predicates, e.g., equality, inequality, band, and non-equality? Third, what other types of join conditions give rise to specialized methods, like RecPart for band-join, so that similarly significant improvements over generic theta-join approaches can be achieved?

More information about RecPart and other techniques for distributed data-intensive computations can be found at https://northeastern-datalab.github.io/distributed/.

ACKNOWLEDGMENTS

This work was supported in part by the National Institutes of Health (NIH) under award number R01 NS091421 and by the National Science Foundation (NSF) under award number CAREER IIS-1762268. The content is solely the responsibility of the authors and does not necessarily represent the official views of NIH or NSF. We also would like to thank Aditya Ghosh for his contributions to the design and implementation of Grid*, Dhaavalkumar Patel for implementing the join-sampling algorithm, and the anonymous reviewers and Niklas Smedemark-Margulies for their constructive feedback.

REFERENCES


