Load Balancing and Skew Resilience for Parallel Joins

Aleksandar Vitorovic, Mohammed Elseidy and Christoph Koch
École Polytechnique Fédérale de Lausanne
Email: {firstname}.{lastname}@epfl.ch

***Abstract*—We address the problem of load balancing for parallel joins. We show that the distribution of input data received and the output data produced by worker machines are both important for performance. As a result, previous work, which optimizes either for input or output, stands ineffective for load balancing. To that end, we propose a multi-stage load-balancing algorithm which considers the properties of *both* input and output data through sampling of the original join matrix. To do this efficiently, we propose a novel category of *equi-weight* histograms. To build them, we exploit state-of-the-art computational geometry algorithms for rectangle tiling. To our knowledge, we are the first to employ tiling algorithms for join load-balancing. In addition, we propose a novel, join-specialized tiling algorithm that has drastically lower time and space complexity than existing algorithms. Experiments show that our scheme outperforms state-of-the-art techniques by up to a factor of 15.**

I. Introduction

There is an increasing demand for scalable and efficient parallel processing of large amounts of data. Load balancing is crucial for reaching this goal, as the total execution time depends on the slowest machine. In this paper, we develop algorithms and techniques for efficient and accurate load balancing for parallel joins.

Join types. The state-of-the-art parallel *equi-joins* rely on hashing with special handling for heavy hitters (join keys with high multiplicity). Examples are the PRPD [1] and F-SkewJoin [2] schemes. Beame *et al.* [3] prove that a modified PRPD scheme [1] is close to the communication optimum.

Unfortunately, these approaches are limited to equi-joins. In contrast, we propose a partitioning scheme for a broad class of *monotonic joins* [4] that include combinations of equality, band and inequality ($<$, $\leq$, $>$, $\geq$) join conditions (e.g., band-join is a combination of 2 inequality join conditions). Still, for joins with only equality conditions, one should use existing approaches (e.g. [3]).

Monotonic joins often arise in practice. Notable examples of band-joins are time-distance joins (e.g. in call logs [5]) and space-distance joins (e.g. in locating nearby objects [4]).

Skew types. Data skew occurs frequently in industrial applications [1], [2]. Load balancing is challenging in the presence of two major types of skew [6]. First, *redistribution skew (RS)* represents uneven input data partitioning among the machines due to skew in the join keys. Thus, RS impedes performance. For instance, the 1-Bucket scheme [4] achieves up to $5\times$ speedup by addressing RS.

Second, *join product skew (JPS)* [6] represents imbalance in load due to variability in the join selectivity, causing disproportionate numbers of output tuples to be processed among parallel workers. That is, a few machines produce a large portion of the output. These machines become bottleneck, severely hindering performance. In fact, JPS can occur even in the absence of RS [7]. The following example illustrates that.

Example 1.1: Let us consider a band join with condition $|R_1.A - R_2.A| \leq 10$. Let us consider the bucket with range $[0..30]$ assuming that each relation has 10 join keys in this range. If $R_1.A$ and $R_2.A$ never satisfy the join condition (e.g. $R1.A$ in $[0..9]$ and $R2.A$ in $[20..29]$), the output size is 0. On the other hand, if each of $R_1.A$ and $R_1.A$ has 10 distinct values in $[0..9]$, the output size is 100.

Although each bucket has the same size $b_s = 10$ (there is no RS), the join output size per bucket varies from 0 to $b_s^2 = 100$ (there is JPS), depending on the the *relative* distribution of the join keys between the input relations. Thus, using only bucket sizes leads to inaccurate estimation of the output distribution, which results in JPS.

Depending on the input and output sizes, JPS may impede performance more than RS. Our evaluation shows that our scheme, which addresses both RS and JPS, achieves up to $15\times$ speedup compared to a state-of-the-art scheme which addresses only RS [4].

Previous work. We present approaches that, similar to ours, go beyond just equi-joins and also support band-joins, inequality-joins etc. We classify previous work as follows. *JPS-avoidance* schemes (e.g. 1-Bucket [4]) balance the output-related work among the machines, regardless of the join selectivity. However, these schemes heavily replicate the input tuples, causing high network and memory consumption, high input-related work per machine, and thus, high execution time. *JPS-susceptible* schemes (e.g. M-Bucket [4]) do not estimate the join output distribution. Hence, these schemes cannot address JPS, causing high output-related work per machine. In general, previous work does not capture the output distribution, as this requires the output sample. Building the sample is hard, as a join between uniform random samples from the input relations is not a uniform random sample of the join output [8].

Our scheme. We propose a novel partitioning scheme which eliminates both RS and JPS. As Table I shows, we are the first to provide a scheme which is *both* input- and output-optimal. In contrast to previous work, our scheme achieves load balancing on *minimal* work per machine, which includes both input- and output-related work. This results in better execution times.

TABLE I: Comparison with most important related work.

Partitioning Scheme	*Input-Optimal*	*Output-Optimal*
1-Bucket	✘	✔
M-Bucket	✔	✘
EWH(ours)	✔	✔

To build such a partitioning scheme [i] , we solve two problems. First, we propose an efficient parallel scheme for capturing the output distribution. We represent the input and output distribution as a matrix, where each dimension of the matrix corresponds to the join keys from an input relation. Second, using these distributions, we optimally assign portions of the matrix (called regions) to machines.

To do so, we introduce a novel family of histograms which we call *equi-weight histograms*, and a novel *histogram algorithm* to build them. An equi-weight histogram is a partitioning of the matrix into regions where regions have almost the same weight (the region weight corresponds to the machine's work). Thus, a partitioning scheme based on the equi-weight histogram *by design* provides for accurate load balancing.

Our histogram algorithm builds on state-of-the-art computational geometry (CG) algorithms for rectangle tiling. To our knowledge, we are the first to employ CG algorithms for join load balancing. Using existing CG algorithms require $\mathcal{O}(n^5 \log n)$ time to produce an accurate partitioning (n is the input relation size). This is impractical, as it is more costly than executing the join itself. In contrast, our algorithm runs in $\mathcal{O}(n)$ time, *while* providing for accurate load balancing, close to that of the baseline CG. We achieve efficiency and accuracy as follows.

First, we devise a novel CG algorithm that employs the domain-specific knowledge about monotonic joins (the properties of the join output distribution). This algorithm drastically reduces the time and space complexity compared to the baseline CG, *while* providing the same accuracy.

Second, we devise a 3-stage histogram algorithm (sampling, coarsening and regionalization), where the output of each stage is the input to the next one in the chain. Each stage reduces the input size of the next one, while providing guarantees for its output. As later stages have more coarse-grained input, we employ more precise algorithms for them to preserve accuracy. As more precise (and expensive) algorithms work on smaller inputs, we preserve efficiency.

Third, we resolve the challenging problem of setting the right output size for each stage. Namely, the size must be small enough to keep the algorithm running time short. On the other hand, the size must be big enough, as insufficient output granularity (resolution) leads to inaccurate load balancing (one machine is assigned much more work than the others).

We explain highlights of our solution while outlining the main contributions of this work:

1. To provide for efficient and accurate load balancing, we devise a multi-stage histogram algorithm which contains a novel, join-specialized computational geometry algorithm.

2. Our scheme achieves *minimal* work per machine, without imposing any assumptions about the data distribution.

[i]By a partitioning scheme we mean either the algorithm for generating the partitioning, or the partitioning itself. In this case we mean the latter. In general, the meaning should be clear from the context.

TABLE II: Summary of the notation used in the paper.

Symbol	*Description*	*Value*
R_1, R_2	Input relations	
J	The number of machines	
n	Max. input relation size	
m	Join output size	
ρ_{oi}	Output/Input ratio	
$w(r)$	Weight of region r	
$\mathcal{M}$	Original join matrix	
$\mathcal{M}_\mathcal{S}$	Sample matrix of size $n_s \times n_s$	$n_s = \sqrt{2nJ}$
s_i	Input sample size	$\Theta(n_s \log n)$
s_o	Output sample size	$\Theta(n_s)$
$\mathcal{M}_\mathcal{C}$	Coarsened matrix of size $n_c \times n_c$	$n_c = \Theta(J)$
$\mathcal{M}_\mathcal{H}$	Equi-weight histogram	

3. We experimentally validate our scheme. Compared to state-of-the-art, our scheme achieves up to $15\times$ speedup in terms of total time (which includes both building the scheme and performing the join) and is up to $5\times$ more efficient in terms of resource consumption.

II. Background & Preliminaries

Next, we introduce definitions used in this paper. Then, we discuss the schemes from Table I, and highlight the benefits of our scheme on a concrete example. Finally, we define the problem statement. Important symbols used in this paper are summarized in Table II.

Join Model. We model a join among relations R_1 and R_2 as a join matrix $\mathcal{M}$. For row i and column j, cell $\mathcal{M}(i,j)$ represents a potential output tuple. $\mathcal{M}(i,j)$ is 1 iff r_i and s_j tuples satisfy the join condition. Figure 1a shows a matrix for a band-join with a join condition $|R_1.A - R_2.A| \leq 1$. We focus on the joins which are common in practice, and for which state-of-the-art techniques perform poorly, that is, *low-selectivity joins*[ii]. These joins have sparse join matrices, i.e. only a small portion of the Cartesian space produces output tuples.

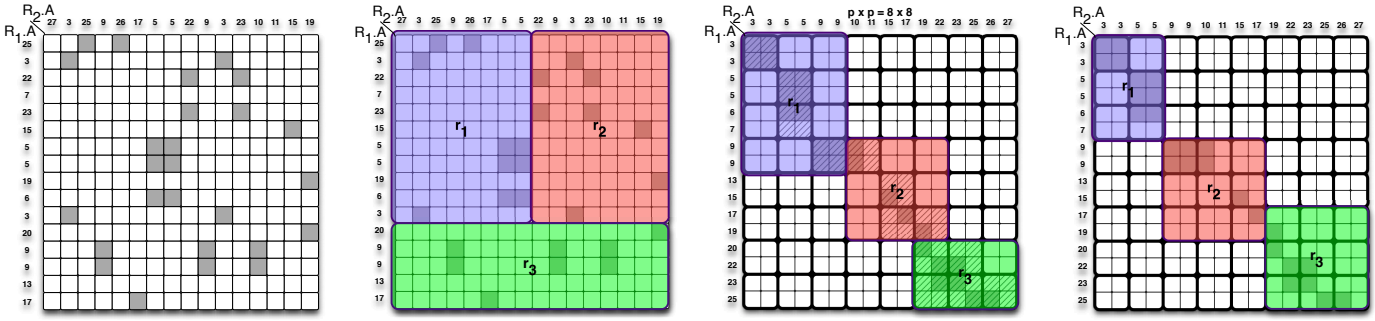
Regions. We execute a join using J machines in a shared-nothing architecture. We refer to a set of cells (that is, the corresponding input tuples) assigned to a single machine for local processing as a *region*. We adhere to rectangular regions, as opposed to rectilinear or non-contiguous regions, to incur minimal storage and communication costs [9].

***Input* and *Output* metrics.** A region's $input$ is its semi-perimeter, that is, the sum of the number of rows and columns from the join matrix intersecting the region. Processing an input tuple consists of receiving the tuple (which incurs network and demarshalling costs) and join computation[iii]. The $output$ is the number of output tuples (frequency) of a region. We use frequency as opposed to area as we focus on *low-selectivity* joins. The processing cost of an output tuple mainly comes from post-processing (writing the output to disk or transferring it over the network to the next operator in the query plan). For example, region r_1 in Figure 1b has $input = 19$ and $output = 10$.

Load balancing is defined as minimizing the maximum work per machine. As each machine is assigned a region, we represent the machine's work as a weight function of $input$ and $output$ costs: $w(r) = c_i(r) + c_o(r)$. As these costs depend on the local join algorithm and hardware/software architecture, $c_i(r)$ and $c_o(r)$ naturally mimic the actual cost of processing

[ii]We also run high-selectivity joins with minimal overhead (see §VI-E).

[iii]The computation cost can partially belong to $output$ (see Section VI-A).



(a) Matrix for a band-join (b) CI(1-Bucket): I_1=19; O_1=10 (c) CS_I(M-Bucket): I_1=14; O_1=14 (d) CS_{IO}(EWH): I_1=10; O_1=10

Fig. 1: Different partitioning schemes (of 3 machines) on a band-join with a join condition $|R_1.A - R_2.A| \leq 1$. Shaded cells represent output tuples. (b)-(d) I_r is *input* and O_r is *output* metric of a region r with maximum weight $w_{x \in 1..J} = I_x + O_x$.

$input(r)$ and $output(r)$ tuples, respectively. Thus, the load-balancing goal can be expressed as minimizing the maximum $w(r)$. Next, we discuss whether different partitioning schemes achieve this goal.

A. Content-Insensitive Partitioning Scheme

The *content-insensitive* partitioning scheme, CI (called 1-Bucket in [4], [9]), illustrated in Figure 1b, assigns all cells (n^2 of them) to machines, regardless of the join condition. Thus, regions cover the entire join matrix. This ensures result completeness and avoids expensive post processing or duplicate elimination. An incoming tuple from R_1 (R_2) randomly picks a row (column) in the join matrix, and is assigned to all the regions which intersect with the row (column). The choice of a row or a column is completely random, and it does not depend on the tuple at all (this is why the scheme is called content-insensitive). For example, the tuple with join key 17 from R_2 randomly picks column 5, which intersects with regions r_1 and r_3. Thus, the tuple is assigned to these regions.

The scheme achieves almost perfect load balancing for *output* by ensuring that regions have almost the same area. In particular, due to random tuple distribution, almost equal-area regions have almost equal *output*.

However, CI incurs prohibitively high *input* costs for *low-selectivity* joins. Namely, as this scheme assigns all the cells (regardless of whether they produce an output tuple) to machines, CI suffers from excessive input tuple replication.

We compare partitioning schemes in Figures 1b, 1c and 1d using the weight function $w(r) = input(r) + output(r)$. Due to excessive tuple replication, CI has the highest maximum $w(r)$ ($w(r_1) = 29$ compared to $w(r_1) = 28$ and $w(r_1) = 20$ of the other schemes). In fact, CI works well *only* if the *output* costs are much bigger than the *input* costs, as in that case input tuple replication has small effect on the work per machine.

B. Content-Sensitive Partitioning Scheme

A *content-sensitive* scheme addresses the excessive tuple replication problem. It assigns an input tuple to a machine(s) according to its content (join key).

CS_I (Figure 1c), called M-Bucket in [4], is a *content-sensitive* scheme that uses the input statistics. To simplify notation, we denote both relation sizes as n [iv]. CS_I builds approximate equi-depth histograms[v] with p buckets over join keys of each input relation ($p < n$), and creates a grid of size $p \times p$ over the join matrix. In Figure 1c, $p = n/2 = 8$, and each grid cell contains $h = (n/p)^2 = 4$ matrix cells. We denote a grid cell which may produce an output tuple as a *candidate cell* (marked with diagonally engraved lines).

To efficiently check if a grid cell is a candidate (in $\mathcal{O}(1)$ time), CS_I requires a join condition that allows candidacy-checking by examining only join keys on the grid cell boundaries. This holds for monotonic joins, as the boundary join keys are sorted. For example, grid cell $(0, 1)$ in Figure 1c is non-candidate, as the distance between the lower R_2 and upper R_1 cell boundary join keys ($5 - 3 = 2$) exceeds the width of the band-join (1).

CS_I optimizes the *input* costs, as it assigns only candidate grid cells to machines, safely disregarding large contiguous portions in the join matrix that produce no output. CS_I scheme assigns tuples to regions according to intersection of the rows and columns with regions. For instance, in Fig. 1c, a tuple from R_1 with join key 9 is forwarded to regions r_1 and r_2. Whereas, all other tuples from R_1 are forwarded to exactly one region. However, as regions are rectangular, CS_I also assigns *some* non-candidates to machines. For example, although grid cell $(0, 1)$ in Figure 1c is non-candidate, it is assigned to r_1. In Figure 1, the maximum $input(r)$ of CS_I is only $I_1 = 14$, compared to $I_3 = 21$ of CI. The gap between the two schemes deepens with increasing the number of machines J, as the number of non-candidates grows.

However, CS_I is susceptible to JPS, as it ignores the actual number of output tuples (*output*) and assigns a constant to each candidate cell. In practice, the *output* of a grid cell varies from 0 to the size of the Cartesian product between the encompassed input tuples from the two relations, that is, from 0 to the grid cell area h ($h = 4$ in our example). In Figure 1c, regions r_1 and r_2 have the same number of candidate cells (4), but vastly different number of output tuples (14 versus 5, respectively). This is why the maximum $w(r)$ in CS_I ($w(r_1) = 28$) only slightly improves that of CI ($w(r_1) = 29$). Thus, JPS prevents CS_I from performing better compared to CI. In fact, CS_I works well *only* if the *input* costs are much bigger than the *output* costs, as in that case JPS marginally affects the work per machine.

[iv] Our analysis also holds when the sizes differ.

[v] For the sake of this example, we assume the exact histogram.

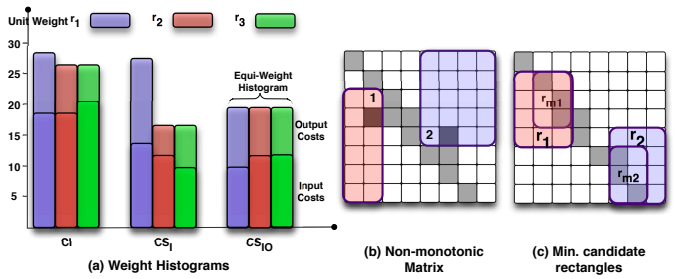


Fig. 2: a) Weight Histograms. b) Non-monotonicity in rectangles 1 and 2 due to candidates marked with black. c) r_{m1} (r_{m2}) is a minimal candidate rectangle for r_1 (r_2).

C. Equi-Weight Histogram Scheme

We propose a novel, equi-weight histogram scheme, CS_{IO} (Figure 1d), which achieves the best of both worlds: it avoids *both* JPS and excessive tuple replication.

CS_{IO} is a *content-sensitive* scheme that accurately estimates the number of output tuples per candidate cell via sampling (see Section IV-A)[vi]. In contrast to CS_I, CS_{IO} is resilient to JPS. This is why the maximum $w(r)$ in CS_{IO} ($w(r_1) = 20$) is much smaller than that of CS_I ($w(r_1) = 28$). In practice, the gap between the two schemes is much deeper. Namely, to have acceptable time for building the CS_I scheme, it must hold that $p \ll n$ [4]. This increases the candidate grid cell area h, making CS_I much more prone to JPS.

An optimal partitioning scheme minimizes the maximum $w(r)$. In contrast to CI and CS_I which work well only if *output* or *input* costs dominate, our CS_{IO} is close-to-optimum for a wide range of *output/input* costs. We build such a scheme using a novel *equi-weight* histogram. The histogram contains buckets of almost equal weight, where each bucket corresponds to a rectangular region. Figure 2 depicts weight histograms for different schemes from Figure 1. CS_{IO} is based on equi-weight histograms and it is the only scheme that minimizes the maximum region weight (machine's work), providing *by design* for accurate load balancing. Next, we formalize the histogram construction problem.

Problem definition. *Given a sparse matrix $\mathcal{M}[1..n, 1..n]$ with cell values {0, 1}, partition it to $J \ll n$ non-overlapping axis-parallel rectangular regions $r_j \in \mathcal{R}$, such that each 1-cell is covered by exactly one region, and each 0-cell is covered by at most one region. The goal is to minimize the maximum weight of a region, that is $\min\{\max_{r_j \in \mathcal{R}}\{w(r_j)\}\}$. $\mathcal{M}$ is the original join matrix where* 1-cells *represent output tuples and* 0-cells *depict empty entries, $w(r)$ represents the work of a machine assigned to region r, and J is the number of joiners (machines).*

The join matrix is just a model. The histogram algorithm does not build $\mathcal{M}$, as this is the actual join result (this would defeat the purpose of building the scheme for parallel join execution). Rather, we resort to sampling (see next section, particularly Section III-A).

III. Histogram Algorithm

In this section, we show how our efficient histogram algorithm achieves accurate load balancing. We first provide a high-level overview of the different stages of our algorithm.

[vi]For the sake of this example, we assume exact statistics.

TABLE III: The time complexity improvements.

BSP over $\mathcal{M}$	*BSP over* $\mathcal{M}_S$	*BSP over* $\mathcal{M}_C$	*MonotonicBSP over* $\mathcal{M}_C$
$\mathcal{O}(n^5 \log n)$	$\mathcal{O}((nJ)^{2.5} \log n)$	$\mathcal{O}(n^{5/3} \log n)$	$\mathcal{O}(n)$

Previous work. The histogram construction problem is NP-hard. The best known approximate algorithm is BSP [10], a tiling algorithm which runs in $\mathcal{O}(n^5)$ time and has an approximation ratio 2.

To create a histogram with J buckets (regions), we need to perform a binary search over the BSP (see §III-C). We denote the entire process as *regionalization*, and it takes $\mathcal{O}(n^5 \log n)$ time. This is impractical, as it is more costly than the join.

Our solution. We propose a histogram algorithm that takes $\mathcal{O}(n)$ time on a single machine, *while* providing for load balancing that is close to the one of the BSP [10]. Our idea is twofold. First, we reduce the input matrix size of the regionalization (originally, regionalization takes the original matrix $\mathcal{M}$ as the input). Second, we drastically improve the regionalization running time by using a novel tiling algorithm which we call MonotonicBSP. These ideas allow us to be the first to use tiling algorithms for join load balancing.

1. Reducing the regionalization input. We introduce the *sampling stage*, which generates sample matrix $\mathcal{M}_S$ of size $n_s \times n_s$. $\mathcal{M}_S$ has much smaller size than the original matrix $\mathcal{M}$ ($n_s \ll n$). To provide for load balancing, n_s needs to be (at least) $\sqrt{2nJ}$ (see §III-A). If the regionalization takes $\mathcal{M}_S$ as the input, it runs in $\mathcal{O}(n_s^5 \log n) = \mathcal{O}((nJ)^{2.5} \log n)$ time. This computation cost is still too high.

To further reduce the regionalization input, we introduce the *coarsening stage*. This stage takes $\mathcal{M}_S$ as the input and creates a coarsened matrix $\mathcal{M}_C$ of size $n_c \times n_c$ ($n_c < n_s$). The coarsening reduces the regionalization input by using the distribution of $\mathcal{M}_S$ cell weights (i.e., we represent multiple small $\mathcal{M}_S$ cells as one $\mathcal{M}_C$ cell). To provide for load balancing, we opt for $n_c = 2J$ (see §III-B). If the regionalization takes $\mathcal{M}_C$ as the input, it runs in $\mathcal{O}(J^5 \log n)$ time. As using $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$ machines is sufficient in practice[vii], the regionalization takes $\mathcal{O}(n^{5/3} \log n)$ time. This is still expensive compared to the join costs.

2. MonotonicBSP: a novel tiling algorithm. In contrast to BSP which takes $\mathcal{O}(J^5 \log n)$ time, our MonotonicBSP runs in only $\mathcal{O}(J^3 \log^2 n)$ time. To do so, MonotonicBSP exploits the output properties of monotonic joins (see §III-C). As $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$ [vii], the regionalization based on MonotonicBSP, along with the sampling and coarsening, takes only $\mathcal{O}(n)$ time (see §III-A to III-C). Table III summarizes all the complexity improvements.

Putting everything together. Figure 3 illustrates the chain of the histogram algorithm stages (the sampling, coarsening and regionalization) for $w(r) = input(r) + output(r)$. The sampling stage builds $\mathcal{M}_S$ of size $n_s \times n_s$ ($n_s = 16$ in Figure 3a) using small input and output samples from $\mathcal{M}$.

[vii]Due to parallelization overhead, adding machines after a certain point provides no additional performance benefits. Our formula for J captures this observation and states that, for example, if n is hundreds of millions, it is then sufficient to use hundreds of machines.

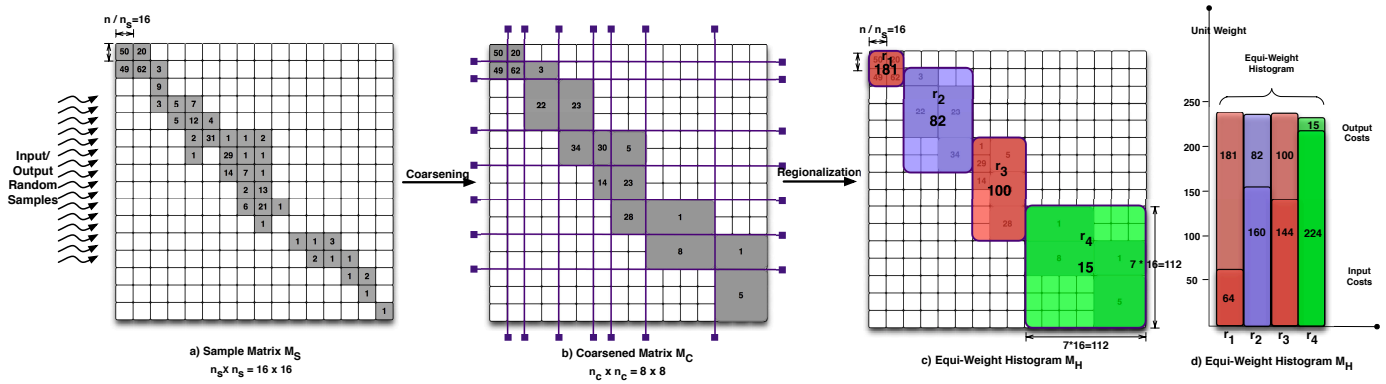


Fig. 3: Histogram algorithm stages. The weight function is $w(r) = c_i(r) + c_o(r) = input(r) + output(r)$. For instance, in (c), $w(r_4) = 2 \cdot 112 + 15 = 239$.

$\mathcal{M}_\mathcal{S}$ preserves the weight distribution of $\mathcal{M}$. That is, with high probability, regions' weights in $\mathcal{M}_\mathcal{S}$ are very close to the corresponding weights in $\mathcal{M}$. The coarsening stage creates a non-uniform grid $n_c \times n_c$ over $\mathcal{M}_\mathcal{S}$ ($n_c = 8$ in Figure 3b), such that each grid cell becomes an $\mathcal{M}_\mathcal{C}$ cell. Thus, $\mathcal{M}_\mathcal{C}$ is of size $n_c \times n_c$. The frequency ($output$) of an $\mathcal{M}_\mathcal{C}$ cell is the sum of the corresponding $\mathcal{M}_\mathcal{S}$ cell frequencies, e.g. $output(\mathcal{M}_\mathcal{C}(1,2)) = output(\mathcal{M}_\mathcal{S}(1, 2..3)) = 3$. The weight is $w(\mathcal{M}_\mathcal{C}(1,2)) = 3n/n_s + output(\mathcal{M}_\mathcal{C}(1,2)) = 3 \cdot 16 + 3 = 51$. The regionalization builds the equi-weight histogram $\mathcal{M}_\mathcal{H}$ by coalescing $\mathcal{M}_\mathcal{C}$ cells into regions (see Figure 3c). This stage uses the hierarchical partitioning (recursively dividing rectangles into 2 sub-rectangles) over its input ($\mathcal{M}_\mathcal{C}$). The hierarchical partitioning allows more configurations that the grid partitioning from the coarsening stage. For example, the grid partitioning cannot produce the hierarchical partitioning from Figure 3c, as r_1 and r_2 partially overlap over the y-axis.

The main ideas in our histogram algorithm that allow for efficient and accurate load balancing are:

1) **We avoid imposing any assumptions** about distribution within a cell, as it leads to incorrect weights and inaccurate load balancing. Thus, we create $\mathcal{M}_\mathcal{C}$ cells (regions) on the granularity of an $\mathcal{M}_\mathcal{S}$ ($\mathcal{M}_\mathcal{C}$) cell.

2) **Careful choice of the matrix sizes** n_s and n_c.

3) **Using more precise algorithms when it matters** for accuracy of load balancing, that is, when the cell weights in the input matrix of the stage are high (i.e., on more coarse-grained matrices). In particular, as we move forward in the chain of the stages, the matrix size drops and the maximum cell weight grows. For example, in Figure 3, the matrix sizes are $n_s = 16$ and $n_c = 8$, and the maximum cell weights are $w(\mathcal{M}_\mathcal{S}(1,1)) = 2 \cdot 16 + 62 = 94$ and $w(\mathcal{M}_\mathcal{C}(6,6)) = 6 \cdot 16 + 8 = 104$. We account for this by using more precise algorithms as we move forward in the chain. Namely, the coarsening considers only grid configurations (of size $n_c \times n_c$) over its input ($\mathcal{M}_\mathcal{S}$), as illustrated in Figure 3b. Whereas, the regionalization is more precise than the coarsening, as it explores all the hierarchical partitionings over its input ($\mathcal{M}_\mathcal{C}$), as illustrated in Figure 3c. More precise algorithms are also more expensive per input matrix cell (see §III-A to III-C). However, as **more expensive algorithms work on smaller input matrices** (recall $n_s = 16$, $n_c = 8$), the histogram algorithm is efficient.

4) Devising **MONOTONICBSP, a novel tiling algorithm**.

5) All the stages use a weight function which accurately estimates the processing costs, and **each stage provides guarantees for minimizing its maximum cell weight**. We prove an upper bound on the $\mathcal{M}_\mathcal{S}$ cell weight (Lemma 3.1). The coarsening (regionalization) guarantees to produce partitionings within a factor of 2 from the optimum on grid (arbitrary[viii]) partitioning on the given $\mathcal{M}_\mathcal{S}$ ($\mathcal{M}_\mathcal{C}$) matrix.

Next, we discuss the details of each stage.

A. Sampling

This stage efficiently builds the sample matrix $\mathcal{M}_\mathcal{S}$, which provides for accurate load balancing.

Region weight proximity. As the histogram algorithm requires precise region weights for accurate load balancing, $\mathcal{M}_\mathcal{S}$ must preserve the region weights of the original matrix $\mathcal{M}$. As we previously saw that regions are defined by their boundary keys, a region r_s from $\mathcal{M}_\mathcal{S}$ *corresponds to* a region r from the original matrix $\mathcal{M}$ if and only if they share all the region boundary keys. The region weight proximity means that for any two corresponding regions r_s and r, $\mathcal{M}_\mathcal{S}$ ensures that $w(r_s) \approx w(r)$ with very high probability. In other words, any region in the sample matrix has almost the same weight as the corresponding region in the original matrix.

Previous work on sample data structures mainly concerns multi-attribute single-relation histograms, which are used for answering range queries (e.g. [11], [12]). As the algorithms for building these data structures consider *only* frequency (output), they cannot preserve the $w(r_s) \approx w(r)$ property. Hence, these algorithms fall short for providing accurate load balancing.

Building sample matrix $\mathcal{M}_\mathcal{S}$. In contrast, we build $\mathcal{M}_\mathcal{S}$ of size $n_s \times n_s$ (in Figure 3a, $n_s = 16$), which keeps the $w(r_s) \approx w(r)$ property by preserving *both* the input and output distribution: *a*) To preserve the input distribution, we build an approximate equi-depth histogram [13] with n_s buckets on each input relation. The histogram boundaries form a grid of size $n_s \times n_s$ over the original matrix. Each such grid cell corresponds to an $\mathcal{M}_\mathcal{S}$ cell. Region's $input$ is a product of the number of $\mathcal{M}_\mathcal{S}$ cells on its semi-perimeter and the expected bucket size n/n_s. For example, in Figure 3a, the region defined by $\mathcal{M}_\mathcal{S}(0..1, 0)$ has $input$ of $3 \cdot n/n_s = 48$. *b*) To preserve the output distribution, we take a uniform random sample of the join output. To do so efficiently, we propose a parallel

[viii] It allows any partitioning into rectangles.

sampling scheme (Section IV-A). Once the sample is in place, we increment the corresponding $\mathcal{M}_S$ cell for each sample output tuple. Region's *output* is a product of the ratio of output sample tuples within the region and the total output size m (we show how to find m in §IV-A). For example, in Figure 3a, region $\mathcal{M}_S(0..1, 0)$ has *output* of $50 + 49 = 99$.

Efficiency and Accuracy Considerations. Setting the $\mathcal{M}_S$ size n_s is crucial for both efficiency and accuracy. As the coarsening takes $\mathcal{M}_S$ as the input, in order to keep the running time of this stage short, n_s must be small enough. On the other hand, decreasing n_s may affect accuracy of the histogram algorithm, and thus, accuracy of load balancing. In particular, if we decrease n_s, the $\mathcal{M}_S$ cells correspond to bigger portions in the original matrix and thus, the $\mathcal{M}_S$ cell weights grow. For example, the maximum cell weight in Figure 3a is $\sigma = 2 \cdot 16 + 62 = 94$, which corresponds to $\mathcal{M}_S(1, 1)$. Assume that $\mathcal{M}_S{}'$ differs from the $\mathcal{M}_S$ in Figure 3a only by replacing $n_s = 16$ by $n'_s = 4$. Then, the maximum cell weight is $\sigma' = 8 \cdot 16 + 201 = 329$, which corresponds to $\mathcal{M}_S(0..3, 0..3)$ and which is much bigger than $sigma = 94$. As we avoid imposing assumptions within an $\mathcal{M}_S$ cell, regions are on the $\mathcal{M}_S$ cell granularity and a region contains at least one $\mathcal{M}_S$ cell. Thus, the maximum region weight in the partitioning scheme with $n'_s = 4$ is at least $\sigma' = 329$. Such a scheme is suboptimal compared to the $\mathcal{M}_\mathcal{H}$ scheme from Figure 3c,d, which has the maximum region weight $w(r_1) = 4 \cdot 16 + 181 = 245$. Thus, small n_s leads to weighty regions, which affects accuracy of load balancing.

It is challenging to choose a value for n_s, as in the sampling stage we do not know the maximum $w(r)$ of the $\mathcal{M}_\mathcal{H}$ partitioning scheme. To that end, we find a lower bound on the maximum $w(r)$ of the *optimum* $\mathcal{M}_\mathcal{H}$ scheme. We denote this lower bound as w_{OPT}, and we compute it by dividing the lower bound on the total join work ($w(\mathcal{M})$, where $input(\mathcal{M}) = 2n$ and $output(\mathcal{M}) = m$) [ix] equally among the machines. In fact, $w(\mathcal{M})$ is a lower bound as it assumes no input tuple replication. To ensure accuracy given that the coarsening and regionalization use approximate algorithms, we require that $\sigma \leq 0.5 w_{OPT}$ (rather than simply $\sigma \leq w_{OPT}$). This holds independently from the join condition and join key distribution if $n_s = \sqrt{2nJ}$, as the following lemma shows. The proofs are in Appendix A.

Lemma 3.1: $n_s = \sqrt{2nJ}$ is the minimum $\mathcal{M}_S$ size such that the maximum cell weight σ in $\mathcal{M}_S$ is at most half of the maximum region weight of the optimum $\mathcal{M}_\mathcal{H}$ partitioning. This holds independently from the join condition and the join key distribution, given that $m \geq n$ [x].

Next, we briefly discuss the sizes of the input and output sample, which are required for building $\mathcal{M}_S$. The detailed analysis is in [14]. The input sample size is $s_i = \Theta(n_s \log n)$. We determine that the output sample size is $s_o = \Theta(n_s)$ using Kolmogorov's statistics [15]. Next, given $n_s = \sqrt{2nJ}$, we prove that the sampling stage has low running time.

Lemma 3.2: The time complexity of the sampling stage is $\mathcal{O}(n_s \log n_s)$. For $n_s = \sqrt{2nJ}$ and $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$ [xi], the time complexity is $\mathcal{O}(n/J)$.

B. Coarsening

This stage creates a coarsened matrix $\mathcal{M}_\mathcal{C}$ by imposing a grid of size $n_c \times n_c$ over the input matrix $\mathcal{M}_S$. As $n_c < n_s$, the coarsening further reduces the regionalization input. Figure 3b shows $\mathcal{M}_\mathcal{C}$ with $n_c = 8$. The goal is to minimize the maximum cell weight in $\mathcal{M}_\mathcal{C}$. This is an NP-hard tiling problem. The best approximate algorithm is the coarsening from [16], which has an approximation ratio 2.

Deciding on n_c. To keep the running time short, while achieving accurate load balancing in the regionalization, we opt for $n_c = 2J$. We discuss how such n_c brings accuracy in §III-D. The following lemma proves low time complexity.

Lemma 3.3: The running time of the coarsening algorithm is $\mathcal{O}((n_s + n_c^2 \log n_s) \cdot n_c \log n_s)$. For $n_c = 2J$ and $J = \mathcal{O}(\sqrt[3]{n/\log^2 n_s})$ [xi], the time complexity becomes $\mathcal{O}(n)$.

Monotonicity is a property of the join output distribution which holds for many interesting joins, including equi-, band- and inequality joins. It states that "if cell (i, j) is not a candidate cell, then either all cells (k, l) with $k \leq i, l \geq j$, or all cells (k, l) with $k \geq i, l \leq j$ are also not candidate cells" [4]. That is, *candidate cells are consecutive per row/column*. All the join matrices in Figure 1 are monotonic, while the one in Figure 2b is not due to the candidate cells marked black.

MonotonicCoarsening. We speed up the coarsening algorithm using monotonicity. The coarsening algorithm iteratively improves the coarsened matrix. To do so, in each iteration it computes the weights of all the $\mathcal{M}_\mathcal{C}$ cells. As the weight of non-candidate cells is 0, it suffices to compute only the weights of the candidate cells. Monotonicity allows skipping the non-candidates for free. Thus, the MonotonicCoarsening considers only the candidate cells. This improves the algorithm's running time in practice, although the complexity does not change asymptotically.

C. Regionalization

This stage creates the equi-weight histogram $\mathcal{M}_\mathcal{H}$, which consists of (at most) J rectangular regions over $\mathcal{M}_\mathcal{C}$ cells. Figure 3c illustrates $\mathcal{M}_\mathcal{H}$ with $J = 4$. The goal is to minimize the maximum region weight δ, while covering with regions all the candidate $\mathcal{M}_\mathcal{C}$ cells. [xii] The best such algorithm is *Binary Space Partition* (BSP) [10], [17]. However, BSP solves a dual problem: given the maximum region weight δ, it minimizes the number of regions. To that end, we perform a binary search over δ until BSP returns a partitioning with the available J regions (machines).

BSP [10], [17] is a tiling algorithm based on dynamic programming. It creates an optimum hierarchical partitioning, which is within a factor of 2 from an optimum arbitrary partitioning. BSP (Algorithm 1) analyzes each rectangle in $\mathcal{M}_\mathcal{C}$ as follows. If the rectangle weight is below the given maximum region weight δ, we cover the rectangle with a single region (line 5). Otherwise, BSP splits the rectangle by a horizontal or vertical line such that the total number of regions used for the two sub-rectangles is minimized (lines

[ix] It is a lower bound as it assumes no input tuple replication.

[x] This typically holds in practice. We discuss the extensions to support the general case for m in [14].

[xi] See footnote vii.

[xii] As regions are rectangular, they cover some non-candidates as well, subject to minimizing δ.

Algorithm 1 BSP.

```
1: function BSP(rectangles)
2:     for each rectangle r in rectangles do
3:         r_m = MINIMALCANDIDATERECTANGLE (r)
4:         if w(r_m) ≤ δ then
5:             r.regions = {r_m}
6:         else
7:             for each splitter in r_m do
8:                 {r_1, r_2} = r_m.split
9:                 splits.add({r_1, r_2})
10:            r.regions = min_{{r_1,r_2}∈splits}(r_1.regions ∪ r_2.regions)
11: end function
```

7-9). We obtain a minimal set of regions for a rectangle by using the previously found splitters for each sub-rectangle. We acquire the final regions by extracting them from the rectangle encompassing the entire $\mathcal{M}_\mathcal{C}$.

Extending BSP to join load balancing. As we do not need to assign non-candidate $\mathcal{M}_\mathcal{C}$ cells to machines, we minimize a rectangle so that no candidate cell is omitted (line 3). We denote such a rectangle as *minimal candidate rectangle*. For example, in Figure 2c, rectangle r_1 (r_2) contains no candidate cells on the left and lower (right and upper) boundaries. Thus, before computing the weights, we minimize r_1 (r_2) to its minimal candidate rectangle r_{m1} (r_{m2}).

As the input for the BSP is of size $n_c \times n_c$, and $n_c = 2J$ (see § III-B), BSP runs in $\mathcal{O}(J^5)$ time. With binary search, it takes $\mathcal{O}(J^5 \log n)$ time. As $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$ [xiii], the regionalization based on BSP runs in $\mathcal{O}(n^{5/3} \log n)$ time. This is expensive compared to the join costs.

BSP also suffers from high space complexity, which is proportional to the total number of rectangles in the input matrix $\mathcal{M}_\mathcal{C}$. As a rectangle is defined by 2 corners, and each corner is defined by 2 $\mathcal{M}_\mathcal{C}$ coordinates, the space complexity is $\mathcal{O}(n_c^4) = \mathcal{O}(J^4)$.

MONOTONICBSP. We propose MONOTONICBSP, a novel tiling algorithm which drastically reduces both time and space complexity of the BSP. To that end, MONOTONICBSP exploits the output distribution properties for monotonic joins. In BSP, enumerating rectangles from $\mathcal{M}_\mathcal{C}$ results in high time and space complexity ($\mathcal{O}(n_c^4)$). However, for monotonic joins, only a small portion of these rectangles are minimal candidates. The main challenge is to enumerate only minimal candidate rectangles without even looking at all the rectangles, as this would require $\mathcal{O}(n_c^4)$ time. To that end, we use the following lemma.

Lemma 3.4: A rectangle is defined by the upper left and the lower right corner. For monotonic joins, each defining corner of a minimal candidate rectangle is a candidate cell, yielding $\mathcal{O}(n_c^2)$ minimal candidate rectangles in total.

Rectangles r_{m1} and r_{m2} in Figure 2c are minimal candidate rectangles, and their defining corners are candidate cells.

Thus, by designating each pair of the candidate cells as the rectangle defining corners, the MONOTONICBSP (Algorithm 2) enumerates all the minimal candidate rectangles (lines 6-13). There are $\mathcal{O}(n_c^2)$ such rectangles, as $\mathcal{M}_\mathcal{C}$ has $\mathcal{O}(n_c)$ candidate cells (we deal with low-selectivity joins). Then, the algorithm sorts the rectangles by their semi-perimeter (line 3), and runs a BSP version which considers only minimal candidate rectangles (lines 14-24).

[xiii] See footnote vii.

Algorithm 2 MONOTONICBSP.

```
1: function MONOTONICBSP
2:     rectangles_m = GENERATECANDIDATERECTANGLES()
3:     Sort rectangles_m according to the semi-perimeter
4:     BSPCANDIDATES(rectangles_m)
5: end function
6: function GENERATECANDIDATERECTANGLES
7:     for x_1 = 1 to n_c do
8:         for y_1 in cand. cell indexes in row x_1 do
9:             for x_2 = x_1 to n_c do
10:                for y_2 in cand. cell indexes in row x_2 do
11:                    rectangles_m.add(x_1, y_1, x_2, y_2)
12:    return rectangles_m
13: end function
14: function BSPCANDIDATES(rectangles_m)
15:    for each rectangle r_m in rectangles_m do
16:        if w(r_m) ≤ δ then
17:            r_m.regions = {r_m}
18:        else
19:            for each splitter in r_m do
20:                {r_1, r_2} = r_m.split
21:                r_m1 = MINIMALCANDIDATERECTANGLE(r_1)
22:                r_m2 = MINIMALCANDIDATERECTANGLE(r_2)
23:                splits.add({r_m1, r_m2})
24:            r_m.regions = min_splits(r_m1.regions ∪ r_m2.regions)
25: end function
```

Lemma 3.5: The regionalization stage based on MONOTONICBSP runs in $\mathcal{O}(n_c^3 \log n_c \log n)$ time. For $n_c = 2J$ and $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$ [xiii], the stage takes $\mathcal{O}(n)$ time.

The space complexity of MONOTONICBSP is $\mathcal{O}(n_c^2)$, as there are n_c^2 minimal candidate rectangles (see Lemma 3.4).

MONOTONICBSP significantly outperforms the baseline BSP for monotonic joins, both in terms of space and time complexity. Namely, MONOTONICBSP requires only $\mathcal{O}(n_c^2)$ space and $\mathcal{O}(n_c^3 \log n_c \log n)$ time. Whereas, the baseline BSP runs in $\mathcal{O}(n_c^4)$ space and $\mathcal{O}(n_c^5 \log n)$ time.

D. Putting it all together

The computation cost. By directly applying previous work [10] (i.e., the regionalization based on BSP over the original matrix $\mathcal{M}$), computing the histogram requires $\mathcal{O}(n^5 \log n)$ time. In contrast, our 3-stage histogram algorithm runs in only $\mathcal{O}(n)$ time.

Theorem 3.1: The time complexity of the histogram algorithm is $\mathcal{O}(n)$.

The proof directly follows from Lemmas 3.2-3.5 (each stage runs in $\mathcal{O}(n)$ time).

The accuracy of load balancing. As in our algorithm the regionalization creates regions on the $\mathcal{M}_\mathcal{C}$ cell granularity, we next discuss how much the coarsening stage affects the accuracy of load balancing. For output-only weight functions, Wang [18] shows that the arbitrary partitioning over a grid

partitioning is within a factor of 4 from the arbitrary partitioning over the original data. Applied to our case, if $n_c \geq J$ and the input matrix of the coarsening stage is the original matrix $\mathcal{M}$ (rather than the sample matrix $\mathcal{M}_S$), the coarsening and regionalization produce a partitioning which is at most a factor of 4 from the one produced by the regionalization alone (this holds only for output-only weight functions). We lessen the factor of 4 by choosing $n_c = 2J$ (rather than $n_c = J$) for the $\mathcal{M}_C$ size.

Sampling minimally affects load balancing, as $\mathcal{M}_S$ with very high probability preserves the weight distribution from $\mathcal{M}$ (Section III-A). Further, we minimize the effect of coarsening to accuracy by ensuring that the maximum cell weight in $\mathcal{M}_S$ is at most half of (rather than equal to) the maximum region weight in the optimum $\mathcal{M}_\mathcal{H}$ partitioning scheme (Lemma 3.1). We provide strong empirical evidences for the accuracy of our equi-weight histogram scheme (see Section VI).

IV. Join operator

In this section, we integrate our partitioning scheme into a join operator. First, we collect the statistics, that is, samples of the input and output tuples (see Section IV-A). Then, using these statistics, we build the equi-weight histogram (see Section III). Finally, we distribute and process the data according to the histogram.

Local Join Algorithm. Each machine processes a region using a local join algorithm. As long as all the machines run the same algorithm, our scheme is orthogonal to the local joins.

Sampling the Input Tuples. As described in §III-A, we need a uniform random sample of size s_i from each relation. We build the input sample in one pass in parallel using Bernoulli sampling [19] with a sampling rate of $q_i = s_i/n$.

A. Sampling the Output Tuples

Chaudhuri *et al.* [8] show that we cannot obtain a uniform random sample of the join output by joining uniform random samples from the input relations. Alternatively, performing the entire join and then sampling from the output defeats the purpose of building the equi-weight histogram. The *Stream-Sample* algorithm [8] provides a uniform random output sample without performing the entire join. However, this is a single-machine algorithm. To make it efficient and scalable, we devise a parallel version of the Stream-Sample algorithm. Next, we discuss efficiency of this algorithm in the context of join load balancing. To our knowledge, we are the first to use random samples of the join output for parallel join load balancing. Then, we describe the baseline and parallel Stream-Sample in detail.

Efficiency. The cost of Parallel Stream-Sample, which mainly comes from scanning the input relations, is small compared to the cost of parallel join. This is due to the following:

1) The benefits of using the collected statistics easily surpass the scanning overhead, both in MapReduce [4], [20], [21] and distributed databases [22]. In both cases, scanning involves repartitioning of the join keys [4], [22]. Our experiments (§VI) also show that scanning pays off, as *JPS affects performance much more than scanning*.

2) The output sample tuples contain only join keys, as we use the samples only for building $\mathcal{M}_S$ (and not for propagating it further in the query plan). This reduces the network traffic.

3) The output sample size is much smaller than the input relation size ($s_o = \Theta(n_s) = \Theta(\sqrt{nJ}) \ll n$).

Stream-Sample. The Stream-Sample [8] works only for equi-joins, but we extend it to work for band- and inequality joins.

First, we introduce the notation. The base relations are R_1 and R_2 and a sample from R_1 is S_1. Given a tuple $t_1 \in R_1$ with a join key $t_1.A$, the *joinable set* of t_1 consists of all the tuples from R_2 which are joinable with t_1. For equi-joins, the joinable set of t_1 comprises of all the tuples from R_2 with $t_1.A$ as the join key. For band- and inequality joins, the joinable set contains all the tuples from R_2 with a join key within a certain distance (specified by the join condition) from $t_1.A$. We denote the joinable set size as $d_2(t_1.A)$, and the ensemble of them as d_2. Using the keys from d_2 with an equality condition yields d_{2equi}. WR (WoR) is sampling With (Without) Replacement.

The Stream-Sample algorithm works as follows. We take a WR weighted sample S_1 of size s_o from R_1, where the weight of $t_1 \in R_1$ is $d_2(t_1.A)$. Then, for each $t_{s_1} \in S_1$, we randomly choose $t_2 \in R_2$ from the joinable set of t_{s_1} and produce an output tuple $t_{s_1} \bowtie t_2$.

Parallelization. We design a parallel, scalable version of Stream-Sample, which runs efficiently on the same number of machines as the join itself. For the ease of presentation, we describe it in terms of MapReduce [23] jobs.

1. We build d_{2equi} from R_2 in a single MapReduce job. To reduce the work, we designate R_2 to be smaller of both relations. To partition the work evenly, we assign the R_2 tuples to the machines according to their join keys and the approximate equi-depth histogram on R_2.

2. In this step, we build d_2 and S_1. We create d_2 as follows: Each reducer obtains a range of sorted join keys from d_{2equi} along with their multiplicities. As mentioned before, $d_2(t_1.A)$ is the sum of multiplicities of the join keys from R_2 which are within a certain distance from $t_1.A$ (according to the join condition). Each time a reducer moves in the sorted key sequence such that the joinable set changes (adding or removing a tuple), a new d_2 key-value pair is created.

We also build a WR weighted sample S_1 from R_1, where weights are based on d_2. To do so, we use a parallel one-pass algorithm for WoR weighted sampling [24] which works as follows: It puts each $t_1 \in R_1$ into a priority queue of size s_o using the priority computed as a function of $d_2(t_1.A)$. According to [24], the precise formula for priority is $r^{(}1/w)$, where $r = random(0, 1)$ and w is the weight, which is in our case $d_2(t_1.A)$. After each reducer produces its Max-Heap reservoir, we merge them into a single reservoir using the same priority function. Finally, we transform S_1 from a WoR to a WR sample using [8].

We build d_2 and S_1 together in a single MapReduce job. We assign the d_{2equi} and R_1 tuples to the machines according to their join keys and the approximate equi-depth histogram on R_1 due to the following reasons: First, d_{2equi} tends to be much smaller than R_1. Secondly, by doing so, we balance the work for computing S_1.

3. Finally, we produce a uniform random output tuple for each tuple $t_{s_1} \in S_1$. As we use the output tuple only for building $\mathcal{M}_S$, it contains only a concatenation of the join keys. This relieves us from choosing uniformly at random an R_2 tuple from the joinable set of t_{s_1}, which would require processing R_2 again. Instead, we randomly choose a join key from the joinable set of t_{s_1}, with probability directly proportional to the key multiplicity. These multiplicities are available in d_{2equi}. As S_1 is typically much smaller than d_{2equi}, we assign S_1 and d_{2equi} to the machines according to their join keys and the partitioning of d_{2equi} from step 1. Thus, we sort S_1 and use a Map-only job for this step.

Synergy. We first build equi-depth histograms on R_1 and R_2. Then, we sample input and output tuples in parallel by sharing mappers (sampling the input requires only one reducer). If an input relation has a predicate which filters out many tuples, we reduce the scanning overhead for the join by materializing the filtered relation in the statistics scan.

Parameters. To build the sample matrix, we need to know m (see §III-A). We obtain m from the Parallel Stream-Sample. In particular, as we iterate over the entire R_1 relation in the step 2 of the algorithm, we compute m as $\sum_{t_1 \in R_1} d_2(t_1.A)$.

B. Discussion and Generalization

System architecture. As [25] shows, systems designed for main-memory parallel processing are very popular nowadays (e.g. Shark-Spark [26], Dremel [27]), mainly because of superior performance compared to the disk-based systems. For that reason, recent parallel joins are main-memory operators [22], [25]. We follow the same reasoning and implement our operator in a main-memory parallel system.

Input relations are not necessarily base relations. Rather, a join may contain selection predicates, or it may consume the output from another join. To support these general joins, we build our scheme for each join (i.e. no reusing among different joins), and report this in the total execution time. The M-Bucket scheme [4] adopts the same approach.

Multi-way joins. Our scheme assumes 2-way joins. As our scheme enhances performance especially when the *output* cost matters a lot (e.g. transferring tuples between operators over the network), a multi-way join can be efficiently executed using a sequence of our 2-way joins. In the future, we plan to extend our approach to support a multi-way join in a single join operator, akin to how [5] extends [4].

V. Related Work

Load balancing is extensively studied both in the context of MapReduce and distributed databases. There has been much work done towards devising efficient join algorithms using the MapReduce framework. The predominant join type in MapReduce is *repartition join* [28], which moves each input tuple over the network. In distributed databases, data is already partitioned among the machines (rather than being stored externally, e.g., on HDFS, as in MapReduce). Thus, some tuples can stay on the same machine. *Broadcast join* replicates one relation on all the machines. This is efficient only if the replicated relation is very small [1]. *Directed join* moves portions of one relation to the corresponding locations of the other relation. It typically requires that one relation is physically partitioned by the join key [28]. This is a limitation when we join a relation with other relations using different join keys [28]. We propose a *novel repartition join*, as repartition joins are the most widely applicable.

1. Equi-joins. Most previous work focuses on equi-joins [29], [28], [1], [2], [3], [22], [25] and partitions the input through some variant of hashing. One should use these techniques for joins that have only equality join conditions.

Next, we discuss why hashing techniques fall short for monotonic joins on an example of a band-join. Namely, hashing scatters neighboring join keys, so that the corresponding tuples from the opposite relation need to be replicated. For a band-join with the width of the band of β, each tuple from the opposite relation goes to $2\beta + 1$ machines ($hash(key - \beta)$, $hash(key-\beta+1)$, ... $hash(key+\beta)$ [xiv]). This implies more input-related work, as well as higher network and memory consumption. The overheads grow proportionally to the width of the band β. Range partitioning avoids this problem, as neighboring join keys are in most cases on the same machine. This leads to less tuple replication, and less overall work compared to hash partitioning.

2. Monotonic joins. In this paper we focus on monotonic joins. State-of-the-art techniques in MapReduce are the 1-Bucket and M-Bucket schemes [4] (for detailed discussion, see §II-A,II-B). In contrast to the 1-Bucket scheme [4], our scheme achieves load balancing on *minimal* work per machine. In contrast to the M-Bucket scheme [4], we address JPS. In distributed databases, Stamos *et al.* [30] present a method that covers the entire join matrix with regions, similarly to the 1-Bucket scheme. This method uses a heuristic model to minimize total communication cost. DeWitt et al. [31] studied band-joins with the goal of minimizing disk accesses.

3. Reliability. Bruno *et al.* [2] introduce the term reliability for equi-joins, arguing that the repartitioning overhead "is more predictable" than the imbalance in load due to JPS. We use a similar argument for monotonic joins: the sampling overhead is more predictable than the imbalance in load when JPS is not addressed. In particular, sampling introduces minimal overhead (up to $0.04\times$, as Section VI-E shows). However, this is negligible compared to the speedups that our scheme achieves by addressing JPS (up to $15\times$, see Section VI-B).

Adaptive load balancing. Adaptive skew handling exist for hash joins (e.g., [32], and for general-purpose MapReduce applications (e.g., [33]). These techniques in general work as follows. When a task becomes idle, it takes over some work from the busiest task. This implies moving the tuples over the network multiple times (first to the "busy", then to the "idle" task), which increases the input-related work. In contrast, we ensure that after building the partitioning scheme, each tuple is repartitioned exactly once. Furthermore, the precise estimation of the remaining time for joins essentially requires equi-weight histograms. In contrast to these adaptive approaches which rely on future load distribution estimation, we present equi-weight histograms that accurately capture workload skew and accordingly fairly partition the work. One could combine the two techniques to reap the benefits of both worlds. In particular, we can use our technique for initial partitioning and for feeding the estimator from [33] in the case of necessity for

[xiv] This is an upper bound on the number of machines, as different hash values can be assigned to a single machine.

task reassignment. By doing so, we could obtain a scheme that adapts to run-time changes (e.g., network problems, machine failures), and that drastically reduces number of task reassignments compared to that of [33] alone.

Work-stealing. Work-stealing (e.g., [34]) is a concept related to adaptive load balancing, but with important differences. Rather than moving the partitions among the machines, it implies dividing the workload into many more partitions than the number of available machines. Each machine pulls a new partition once it finishes processing the previous one. However, increasing the number of partitions inherently increases replication. For example, if we divide a partition into two sub-partitions, the corresponding tuples from the opposite relation need to be duplicated. Thus, work-stealing increases the input-related work. Finally, it is not clear how to decide on the number of partitions so that work-stealing avoids JPS.

Sample data structures. The closest data structure to our sample matrix $\mathcal{M}_S$ is single-relation multi-attribute histogram [11], [12], [35], [36], which is used for selectivity estimation. These histograms represent the frequency distribution over a multi-dimensional space. In our $\mathcal{M}_S$, frequency is the number of output tuples for the corresponding segments of the input relations. However, the goal of multi-attribute histograms differs from ours as their aim is to minimize the total frequency errors over the entire domain, rather than to lend support for load balancing. Namely, multi-attribute histograms cannot provide for load balancing, as they capture the frequency rather than the weight distribution, and they cannot guarantee the maximum cell weight nor decide on the $\mathcal{M}_S$ size n_s. Finally, we take advantage of join peculiarities, that is, monotonicity.

VI. Evaluation

This section compares our operator with state-of-the-art operators. We first evaluate the execution time and resource consumption, i.e., memory requirements and network communication. Then, we assess the scalability of each operator. Further, we evaluate the accuracy of our partitioning scheme, along with the efficiency of building it. Finally, we analyze worst-case scenarios for our operator.

A. Experimental Setup

Environment. We perform our experiments on an Oracle Blade 6000 server with 10 Oracle X6270 M2 blades. Each blade has two 3Ghz 6-core Intel Xeon X5675 CPUs. Out of 120 cores, 64 are available exclusively for our experiments. Each blade runs Ubuntu 12.04 and has 72GB of DDR3 RAM and a 1Gbit Ethernet interface. Later on, by a machine assigned to an operator, we mean a core with an exclusively assigned portion of the blade main memory.

Datasets. We run joins over both TPC-H [37] and a synthetic dataset X. We employ the TPC-H generator [38], which creates datasets with *Zipf* distributions set through the skew parameter z. We set $z = 0.25$ to demonstrate that JPS can be large even if RS is moderate. The X dataset has 2 independently generated relations (R_1 and R_2), each with 2 segments. The second and first segment sizes are in proportion 80/20, and joining smaller segments from R_1 and R_2 produces majority of the output. In particular, in the first segment, we generate x tuples and choose its join keys uniformly at random from the $[0..x/6]$ domain. In the second segment, we generate $y = 4 \cdot x$ tuples and choose its join keys uniformly at random from the $[2y, 6y]$ domain. The segments from different relations are independently generated.

TABLE IV: Joins' characteristics. $Input$ and $output$ sizes are in millions of tuples. β is the width of the band.

Name	Dataset	Join condition	$input$	$output$
B^*_{ICD}	TPC-H	Band-join($\beta = 2$)	480M	296M
B_{CB}	X	Band-join($\beta = 1$)	192M	348M
B_{CB}	X	Band-join($\beta = 2$)	192M	580M
B_{CB}	X	Band-join($\beta = 3$)	192M	812M
B_{CB}	X	Band-join($\beta = 4$)	192M	1044M
B_{CB}	X	Band-join($\beta = 8$)	192M	1972M
B_{CB}	X	Band-join($\beta = 16$)	192M	3828M
BE^*_{OCD}	TPC-H	Band/Equi-join($\beta = 2$)	36.8M	2000M

* For joins over the TPC-H data, the database size is $160G$ and $z = 0.25$.

Operators. We evaluate three different operator partitioning schemes: (i) CI (1-Bucket scheme) [4], (ii) CS_I (M-Bucket scheme) [4], and finally, (iii) CS_{IO}, which is our equi-weight histogram scheme.

Configuration. We run the join queries on $J = 32$ machines, whereas for the scalability experiments we use 16 to 64 machines. For the joins over the TPC-H data, we set the scale factor to 160 (i.e., 160GBs) and for the scalability experiments, we set it between 80 and 320. In the histogram algorithm of CS_I[xv], we set the number of buckets p to 2000. In the scalability experiments, we scale p proportionally to J.

Programming model. We use our MapReduce-like system called SQUALL[xvi]. SQUALL is an in-memory system[xvii] which is based on Twitter's STORM[xviii]. The system runs in Java v1.7. Mappers shuffle the input tuples according to the partitioning scheme of the operator. Reducers perform the actual join and randomly shuffle the output tuples to the mappers of the next stage (e.g. join, aggregation). Thus, each mapper performs the same amount of work and it suffices to balance the load among the reducers of a job. The job execution time includes sending the tuples over the network to the next stage.

Cost model. For our experiments we define the weight function (§II) for load balancing among the reducers as[xix]:

$$w(r) = c_i(r) + c_o(r) = w_i \cdot input(r) + w_o \cdot output(r)$$

where w_i, w_o is the average time cost of processing a single input and output tuple, respectively. We determine the values for w_i and w_o using linear regression on several benchmark runs. The regression method automatically divides all the communication and computation costs into c_i and c_o costs. The results of regression in our system suggest the values $w_i = 1$ and $w_o = 0.2$ for band-joins and $w_i = 1$ and $w_o = 0.3$ for combinations of equi- and band-joins.

Joins. As the operator performance is highly correlated to the $output/input$ cost ratio (c_o/c_i), we classify joins to *input-cost dominated (ICD)*, *cost-balanced (CB)* and *output-cost dominated (OCD)*. We evaluate a band-join (B_{ICD}) and a join with band and equality join conditions (BE_{OCD}) over the TPC-H dataset, and a band-join (B_{CB}) with 6 different widths of the band (β) over the X dataset. Table IV summarizes the joins' characteristics. B_{ICD} is an input-cost dominated join,

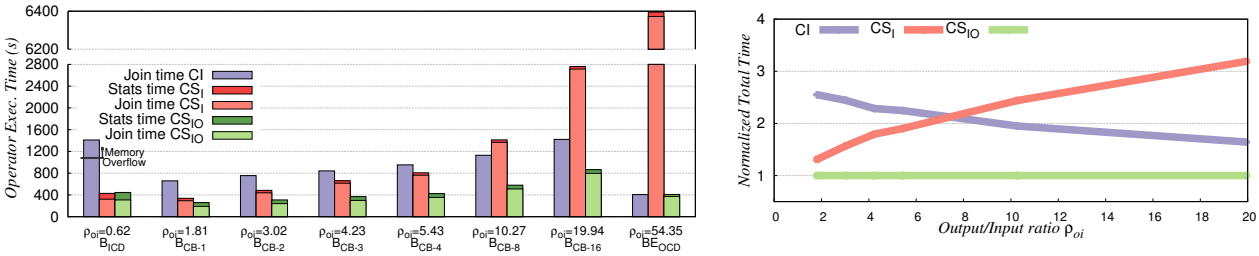
[xv] This histogram algorithm does not create equi-weight histograms; it is a heuristic that builds a partitioning scheme.

[xvi] https://github.com/epfldata/squall/

[xvii] See §IV-B for a discussion about alternative architectures.

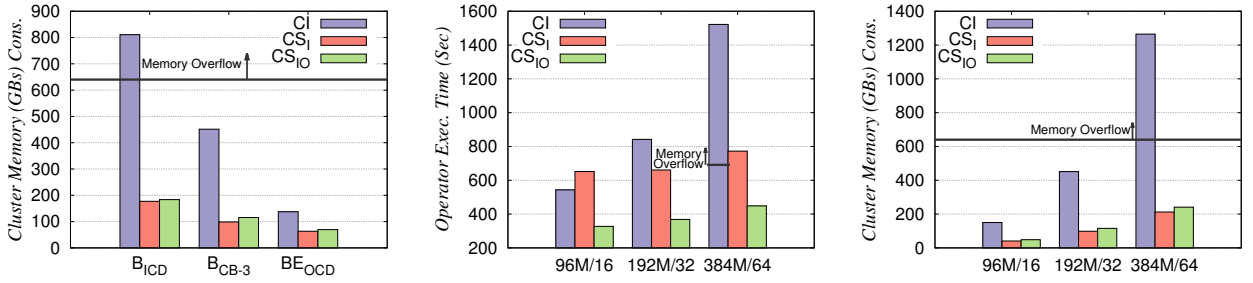
[xviii] http://storm.apache.org/

[xix] The model can be flexibly adapted to represent any realistic cost function.



(a) Total Execution Time.

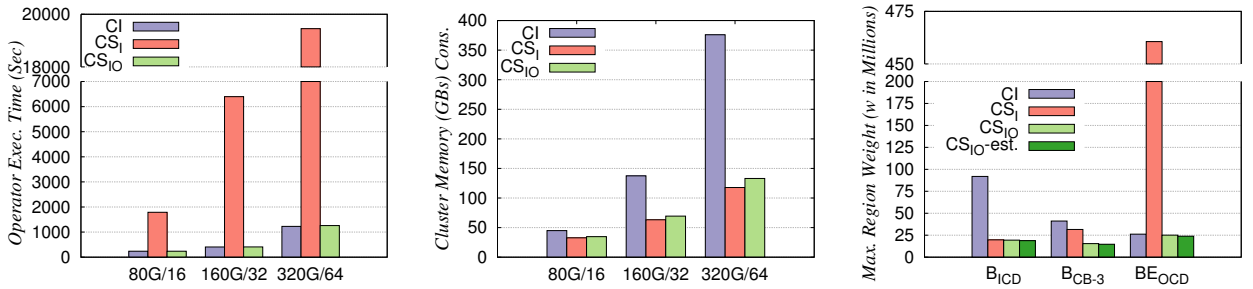
(b) Normalized Execution Time $B_{CB-\beta}$.



(c) Cluster Memory Consumption.

(d) B_{CB-3} Scalability Execution Time.

(e) B_{CB-3} Scalability Memory Cons.



(f) BE_{OCD} Scalability Execution Time.

(g) BE_{OCD} Scalability Memory Cons.

(h) Maximum Region Weight.

Fig. 4: Performance Evaluation

as $c_i = 8.1 \cdot c_o$. BE_{OCD} is an output-cost dominated join, as $c_i = 0.06c_o$. Finally, B_{CB} is cost-balanced, as the *input* and *output* cost are comparable ($0.25c_o \leq c_i \leq 2.75c_o$, depending on β). The joins are defined in Appendix B.

B. Performance Analysis

In this section, we evaluate the operators' performance in terms of the execution time and resource consumption, i.e., memory requirements and network communication. We show that the execution time mainly depends on the join *output*/*input* ratio, which we call ρ_{oi}.

Total execution time is shown in Figure 4a as the sum of the time for building the partitioning scheme ("stats time") and the join execution time ("join time"). CI has only "join time" as it has no preprocessing phase. Figure 4b shows the normalized total execution times for B_{CB}. The operators' execution times are highly correlated to the *output*/*input* ratio: (i) For small ρ_{oi}, i.e., on one side of the spectrum, *input* costs dominate the join execution time. Thus, CI, which replicates each input tuple to 6 machines, performs poorly for B_{ICD}. CS_I avoids this problem, but due to lack of output statistics it suffers from JPS. (ii) For high ρ_{oi}, i.e., on the other side of the spectrum, *output* costs dominate the join execution time. Thus, for BE_{OCD}, the effect of JPS on the join execution time of CS_I escalates, causing CS_I to perform poorly. CI avoids this problem, but it still suffers from high input replication. (iii) As B_{CB} is a cost-balanced join, both existing operators perform poorly. Increasing the width of the band β in B_{CB} leads to the increase in ρ_{oi}, such that the *output* costs grow relatively to *input* costs. This improves the performance of CI and degrades the performance of CS_I compared to CS_{IO}.

Our CS_{IO} outperforms the other operators as it is close-to-optimum on the *total* work per machine, which includes both *input* and *output* costs. CS_{IO} captures the output distribution *and* avoids high input tuple replication. Thus, in terms of the join execution time, CS_{IO} achieves from $1.04\times$ (B_{ICD}) to $17.22\times$ (BE_{OCD}) speedup compared to CS_I, and from $1.1\times$ (BE_{OCD}) to $4.59\times$ (B_{ICD}) speedup compared to CI. Similarly, in terms of the total execution time, CS_{IO} achieves up to $15.63\times$ (BE_{OCD}) speedup compared to CS_I, and up to $3.16\times$ (B_{ICD}) speedup compared to CI. In fact, as CI for B_{ICD} runs out of memory, we extrapolate its total execution time using the cost model parameters and the percentages of processed input and output tuples.

All the SQUALL operators will be faster once we migrate from STORM to Twitter HERON, as HERON is an order-of-magnitude faster[xx] than STORM. HERON is API-compatible with STORM. At the time of preparing this paper, HERON

[xx] http://www.infoq.com/news/2015/06/twitter-storm-heron

TABLE V: Join execution and histogram algorithm time (s) of CS_I for different number of buckets p.

Join	Time	Number of buckets p					
		2000	4000	8000	10000	16000	24000
BE_{OCD}	Join execution	6372	6306	5480	5080	4294	3410
	Histogram alg.	0.4	1.3	5	8.1	19	49
B_{CB}	Join execution	615	604	601	582	569	575
$\beta = 3$	Histogram alg.	0.4	1.4	4.9	6.7	15	36

was not open source yet. In the meantime, HERON became an open source project, and we are currently porting SQUALL to HERON. Once we complete the porting, SQUALL will be on par with other parallel main-memory database systems (e.g. Track Join [22]).

Choosing among CS_I and CI. An important difficulty when using the existing CS_I and CI schemes is that we cannot always choose the better one without knowing the input/output sizes. However, output size estimation using the precomputed statistics is error-prone due to possible predicate correlations [39]. In fact, the estimate can be orders of magnitude away [40]. Output size estimation is even harder for non-equi joins. *This might lead to choosing the worst operator among CI and CS_I*(e.g. CI for an input-cost dominated join). In that case, our CS_{IO} achieves from $2.25\times$ (B_{CB-4}) to $15.63\times$ (BE_{OCD}) speedup.

More detailed input statistics in CS_I (increased number of buckets p in CS_I) cannot cure the lack of output statistics nor address JPS. In particular, Table V shows that for both BE_{OCD} and B_{CB-3}, increasing p leads to increased histogram algorithm time, and thus, increased time for building the partitioning scheme. Increasing p also decreases the join execution time. However, even if CS_I is given more time for building the partitioning scheme than CS_{IO}, its total execution time is still much worse than that of CS_{IO}. For instance, for BE_{OCD} and $p = 24000$, the histogram algorithm grows to 49 seconds, making the time for building the partitioning scheme $1.8\times$ worse than that of CS_{IO}. Moreover, in that case, the total execution time of CS_I is $8.51\times$ worse than that of CS_{IO}.

Resource consumption. Figure 4c illustrates resource consumption, which includes cluster memory and network traffic (input data sent from mappers to reducers). Resource consumption is important because it directly translates to energy consumption and, in cloud environments, to dollar costs. In general, CI has better execution times than CS_I, but it is very resource-inefficient compared to both CS_I and CS_{IO}.

For B_{ICD} and B_{CB-3}, CI requires around $4\times$ more memory than CS_I and CS_{IO}. This is due to the fact that CI excessively replicates tuples (the replication factor is 6, as the partitioning scheme is 4×8 so one relation is replicated $4\times$ and the other is replicated $8\times$). CI requires $4\times$ rather than $6\times$ more memory as both CS_I and CS_{IO} replicate some input tuples (see Figures 1c, 1d). CS_{IO} uses slightly more memory than CS_I, because CS_{IO} balances on the total work, so it assigns more *input* for the regions with relatively small *output*. In fact, as CI for B_{ICD} runs out of memory, we extrapolate its memory consumption using the percentage of the processed input tuples. CI in BE_{OCD} does not have high memory consumption, as the input size is smaller than for B_{ICD} and B_{CB-3}.

C. Scalability

Next, we evaluate the weak scalability of the operators by scaling the data size and the number of machines evenly. We show that our CS_{IO}, in contrast to CS_I and CI, scales well both in the total execution time and resource consumption.

B_{CB-3} total execution time is shown in Figure 4d. We evaluate various *input*/*output*/J settings, more specifically, $96M$/$406M$/16, $192M$/$811M$/32 and $384M$/$1.62B$/64, where M and B stand for millions and billions of tuples. CI scales worse than CS_I and CS_{IO}. This is expected, as the replication factor grows from 4 ($J = 16$) to 8 ($J = 64$), which doubles the *input* costs on each machine. Namely, for $J = 16$ and $J = 64$, CI has $1.66\times$ and $3.39\times$ worse total execution time than CS_{IO}. In fact, as CI with $J = 64$ runs out of memory, we extrapolate its total execution time and memory consumption.

B_{CB-3} resource consumption. Figure 4e shows the cluster memory consumption for B_{CB-3}. As the replication factor grows from 4 ($J = 16$) to 8 ($J = 64$), CI requires increasingly more memory compared to CS_I and CS_{IO}. Namely, for $J = 16$ and $J = 64$, CI requires $3.1\times$ and $5.25\times$ more memory than CS_{IO}, respectively. As CI with $J = 64$ runs out of memory, we extrapolate its memory consumption.

BE_{OCD} total execution time is shown in Figure 4f. The *input*/*output*/J settings are $21.2M$/$612M$/16, $36.8M$/$2B$/32 and $62M$/$8.8B$/64. Taking into account that increasing J from 16 to 64 ($4\times$) causes the output size to grow $14.46\times$, CS_{IO} and CI achieve good scalability. For CS_{IO}, this validates the efficiency of our scheme even for highly output-cost dominated joins (for $J = 64$, $\rho_{oi} = 142.57$). For CI, this is due to the fact that the *output* cost outweighs the *input* cost. On the other hand, CS_I scales very poorly as JPS causes that only few machines produce most of the output. Namely, for $J = 16$, $J = 32$ and $J = 64$, CS_I has $7.52\times$, $15.63\times$ and $15.43\times$ longer total execution time than CS_{IO}, respectively.

BE_{OCD} resource consumption. Figure 4g shows the cluster memory consumption for BE_{OCD}. The gap between CI and the other two operators is smaller than in B_{CB-3} due to the following. As the number of machines J grows from $J = 16$ to $J = 64$ ($4\times$), the input size grows only $2.92\times$. Thus, for $J = 64$, CI takes $2.82\times$ more memory than CS_{IO}.

Scalability summary. Overall, in terms of the total execution time and resource consumption, only CS_{IO} scales well for both B_{CB-3} and BE_{OCD}.

D. Accuracy and Efficiency of CS_{IO}

This section evaluates the accuracy of our partitioning scheme, as well as the efficiency of building it. Building the partitioning scheme consists of collecting the input and output samples and running our 3-stage histogram algorithm.

Accuracy of the cost model. Our cost model represents the join work of a machine as the weight of the region assigned to it. The cost model is accurate if the region weight corresponds to the machine's work, and consequently, if the maximum region weight corresponds to the join execution time. Figure 4h validates the model accuracy as for each join among B_{ICD}, B_{CB-3} and BE_{OCD}, the maximum region weights of different schemes are proportional to the corresponding join execution times. The proportionality holds only within

the same join, as a weight unit represents different amount of work in different joins. We obtain the weights after the join execution by computing the weight function on the number of input and output tuples processed per machine.

Accuracy of our CS_{IO}. Figure 4h shows that the estimated maximum region weight in our histogram algorithm, marked as CS_{IO}-EST., is at most 6% off the value computed after the execution. Thus, our scheme is accurate.

The time for building the partitioning scheme is illustrated in Figure 4a as "stats time". It includes the time to collect statistics (the input statistics for CS_I and input and output statistics for CS_{IO}), and the running time of the histogram algorithm. We evaluate the efficiency of building our partitioning scheme, and compare it to that of CS_I.

Figure 4a shows that building the CS_{IO} scheme takes at most 31% of its total execution time (B_{ICD}). Further, building the CS_{IO} scheme is at most 6.7% more expensive than that of CS_I in terms of the CS_{IO} total execution time (B_{ICD}). This is due to the following reasons: (i) Collecting the input statistics is much cheaper in CS_{IO} than in CS_I. CS_I requires 2 MapReduce stages, while CS_{IO} requires only 1 MapReduce stage. This is due to the fact that CS_I needs to use more buckets than CS_{IO} to account for the error caused by the lack of the output statistics. In the worst case (high JPS), the required number of buckets for CS_I is $\Theta(n)$. In contrast, the number of buckets for CS_{IO} does not depend on skew at all, and it slowly grows with the increase in n ($O(\sqrt{n})$). (ii) The time to collect output statistics for CS_{IO} is not much longer than the second pass of collecting input statistics for CS_I. In addition to a scan over the input relations, which is required by both operators for building the partitioning scheme, CS_{IO} performs a scan over d_{2equi} (step 2 in Section IV-A) and produces the output sample (step 3 in Section IV-A). The former is cheap as d_{2equi} tends to be much smaller than its originating relation, which is the smaller one. The latter is cheap as the output sample size is very small compared to n ($s_o = \Theta(\sqrt{nJ})$).

Accuracy/Efficiency summary. Overall, our equi-weight histogram scheme is practical, as it provides for both accurate and efficient load balancing.

E. Worst-case scenarios

Input-cost dominated joins (small ρ_{oi})/no skew. For very small ρ_{oi} (B_{ICD}), CS_{IO} achieves minimal speedups in the join execution time compared to CS_I ($1.04\times$). This is because the *output* cost is $8.1\times$ smaller than the *input* cost, so JPS minimally affects the performance. In fact, joins with very small ρ_{oi} behave almost as if there was no JPS at all. Thus, in the worst case (B_{ICD}), the total execution time of CS_{IO} is $1.04\times$ higher than that of CS_I.

High-selectivity joins (very high ρ_{oi}). Our scheme is designed for low-selectivity joins. CS_{IO} is better or on par with CI if the output is up to 2 orders of magnitude bigger than the input. We address high-selectivity joins as follows. As we saw in §VI-B, we cannot know join selectivity beforehand. Rather, we take advantage of the following fact. As ρ_{oi} grows, building CS_{IO} requires less time compared to the total execution time of the better among CI and CS_{IO} (from 31% for B_{ICD} to only 9.8% for BE_{OCD}, see Figure 4a). As building the CS_{IO} scheme is comparably cheap for high-selectivity joins, we always start with our scheme, and fall back to CI if needed. We decide when to switch to CI using the following fact. As ρ_{oi} grows, the time for building the CS_{IO} scheme grows relatively to the input sizes (138s for 480M tuples of B_{ICD} to 40s for only 37M tuples of BE_{OCD}). If the time for building the scheme exceeds an experimentally-found threshold (e.g. half a second for each million of input tuples in our setup), we fall back to CI. In that case, we waste only 4% of the total execution time of CI before switching to CI.

Summary. Possible slowdowns (up to $1.04\times$) are negligible compared to the achieved speedups (up to $15\times$).

F. Summary

Joins are defined in a spectrum of cost distribution. At each end, either *input* or *output* costs dominate the join cost. Previous work, that is, CS_I and CI, perform well only at the extreme ends of the $output/input$ spectrum. This is because CI suffers from excessive input tuple replication (which worsens with the increase in the number of joiners), while CS_I cannot capture the *output* cost distribution. Due to errors in the output size estimation, especially for non-equi joins, choosing the wrong operator among CS_I and CI becomes plausible, causing severe performance degradations. In contrast to previous work, our CS_{IO} captures the output distribution, and avoids high input tuple replication. Thus, our scheme is close-to-optimum on the *total* work per machine, which includes both *input* and *output* costs. Consequently, our scheme performs very well over a wide spectrum of $output/input$ ratios, and it scales with increasing data sizes.

CS_{IO} achieves up to $5.25\times$ improvement in resource consumption and up to $3.39\times$ speedup compared to CI. Moreover, CS_{IO} achieves up to $15.63\times$ speedup compared to CS_I. As these speedups refer to the total execution time, they also validate the efficiency of building our scheme, which consists of collecting the input and output samples and running our 3-stage histogram algorithm.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments. We also owe many thanks to Yannis Klonatos. His comments greatly improved the presentation of the paper, especially in describing the algorithms.

REFERENCES

[1] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, "Handling data skew in parallel joins in shared-nothing systems," in *SIGMOD*, 2008.

[2] N. Bruno, Y. Kwon, and M.-C. Wu, "Advanced join strategies for large-scale distributed computation," in *VLDB*, 2014.

[3] P. Beame, P. Koutris, and D. Suciu, "Skew in parallel query processing," in *PODS*, 2014.

[4] A. Okcan and M. Riedewald, "Processing theta-joins using MapReduce," in *SIGMOD*, 2011.

[5] X. Zhang, L. Chen, and M. Wang, "Efficient multi-way theta-join processing using MapReduce," *VLDBJ*, vol. 5, no. 11, 2012.

[6] C. Walton, A. Dale, and R. Jenevein, "A taxonomy and performance model of data skew effects in parallel joins," in *VLDB*, 1991.

[7] V. Poosala and Y. E. Ioannidis, "Estimation of query-result distribution and its application in parallel-join load balancing," in *VLDB*, 1996.

[8] S. Chaudhuri, R. Motwani, and V. Narasayya, "On random sampling over joins," in *SIGMOD*, 1999.

[9] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch, "Scalable and adaptive online joins," in *VLDB*, 2014.

[10] P. Berman, B. DasGupta, and S. Muthukrishnan, "Slice and dice: A simple, improved approximate tiling recipe," in *SODA*, 2002.

[11] M. Muralikrishna and D. J. DeWitt, "Equi-depth multidimensional histograms," in *SIGMOD*, 1988.

[12] V. Poosala and Y. E. Ioannidis, "Selectivity estimation without the attribute value independence assumption," in *VLDB*, 1997.

[13] S. Chaudhuri, R. Motwani, and V. Narasayya, "Random sampling for histogram construction: How much is enough?" in *SIGMOD*, 1998.

[14] A. Vitorovic, M. Elseidy, and C. Koch, "Load balancing and skew resilience for parallel joins," EPFL, Technical Report 203656, 2015.

[15] J. D. Gibbons, *Nonparametric methods for quantitative analysis*. New York: Holt, Rinehart and Winston, 1976.

[16] S. Muthukrishnan and T. Suel, "Approximation algorithms for array partitioning problems," *J. Algorithms*, vol. 54, no. 1, 2005.

[17] S. Muthukrishnan, V. Poosala, and T. Suel, "On rectangular partitionings in two dimensions: Algorithms, complexity, and applications," in *ICDT*, 1999.

[18] Y. Wang, "Relations between two common types of rectangular tilings." *Int. J. Comput. Geometry Appl.*, vol. 19, 2009.

[19] R. Gemulla, P. J. Haas, and W. Lehner, "Non-uniformity issues and workarounds in bounded-size sampling," *VLDBJ*, vol. 22, no. 6, 2013.

[20] C. Doulkeridis and K. Nørvag, "A survey of large-scale analytical query processing in mapreduce," *VLDBJ*, vol. 23, no. 3, 2014.

[21] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "A study of skew in mapreduce applications," in *Open Cirrus Summit*, 2011.

[22] O. Polychroniou, R. Sen, and K. A. Ross, "Track join: distributed joins with minimal network traffic," in *SIGMOD*, 2014.

[23] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI*, 2004.

[24] P. S. Efraimidis and P. G. Spirakis, "Weighted random sampling with a reservoir," *Inf. Process. Lett.*, vol. 97, no. 5, 2006.

[25] S. Chu, M. Balazinska, and D. Suciu, "From theory to practice: Efficient join query evaluation in a parallel database system," in *SIGMOD*, 2015.

[26] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *SIGMOD*, 2013.

[27] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," in *VLDB*, 2010.

[28] S. Blanas, J. Patel, V. Ercegovac, J. Rao, E. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in MapReduce," in *SIGMOD*, 2010.

[29] F. Afrati and J. Ullman, "Optimizing joins in a MapReduce environment," in *EDBT*, 2010.

[30] J. Stamos and H. Young, "A symmetric fragment and replicate algorithm for distributed joins," *Transactions on Parallel and Distributed Systems*, vol. 4, no. 12, 1993.

[31] D. J. DeWitt, J. F. Naughton, and D. A. Schneider, "An evaluation of non-equijoin algorithms," in *VLDB*, 1991.

[32] L. Harada and M. Kitsuregawa, "Dynamic join product skew handling for hash-joins in shared-nothing database systems," in *DASFAA*, 1995.

[33] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *SIGMOD*, 2012.

[34] K. Agrawal, C. E. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[35] A. Aboulnaga and S. Chaudhuri, "Self-tuning histograms: Building histograms without looking at data," in *SIGMOD*, 1999.

[36] N. Bruno, S. Chaudhuri, and L. Gravano, "STHoles: A multidimensional workload-aware histogram," in *SIGMOD*, 2001.

[37] "The TPC-H benchmark," `http://www.tpc.org/tpch/`.

[38] S. Chaudhuri and V. Narasayya, "TPC-D data generation with skew."

[39] Y. Ioannidis and S. Christodoulakis, "On the propagation of errors in the size of join results," in *SIGMOD*, 1991.

[40] M. Stillger, G. Lohman, V. Markl, and M. Kandil, "LEO - DB2's learning optimizer," in *VLDB*, 2001.

[41] P. Berman, B. Dasgupta, S. Muthukrishnan, and S. Ramaswami, "Improved approximation algorithms for rectangle tiling and packing," in *SODA*, 2001.

APPENDIX

A. The histogram algorithm: Details and proofs

1) Sampling: The following lemmas require that $w(r)$ is *monotonic* [xxi] (a region weighs at least the weight of any of its subregions) and that $c_i(r)$ and $c_o(r)$ are *superadditive*, that is, processing $x+y$ input (output) tuples is at least as expensive as the sum of processing costs for x and y input (output) tuples. This holds for realistic cost models.

Lemma 3.1: $n_s = \sqrt{2nJ}$ is the minimum $\mathcal{M}_S$ size such that the maximum cell weight σ in $\mathcal{M}_S$ is at most half of the maximum region weight of the optimum $\mathcal{M}_\mathcal{H}$ partitioning. This holds independently from the join condition and the join key distribution, given that $m \geq n$ [xxii].

Proof: An $\mathcal{M}_S$ cell corresponds to a region in the original matrix with dimensions $n/n_s \times n/n_s$, where n_s is the number of buckets in the equi-depth histogram. Due to the fact that $n_s = \sqrt{2nJ}$, the semi-perimeter of each cell is $\max_{cell}(s_p(cell)) = 2n/n_s = \sqrt{2n/J}$. The maximum frequency of an $\mathcal{M}_S$ cell is given by the Cartesian product between the encompassed input tuples from the two relations. That is, $\max_{cell}(f(cell)) \leq (n/n_s)^2 = n/2J$. Because $m \geq n$, it follows that $\max_{cell}(f(cell)) \leq m/2J$. It holds that $\sigma = \max_{cell}(w(cell)) \leq c_i(\sqrt{2n/J}) + c_o(m/2J)$. As $J \ll n$ (it suffices that $J < n/3$), it follows that $\sqrt{2n/J} < n/J$ and $\sigma \leq c_i(n/J) + c_o(m/2J)$. We denote the maximum region weight of the $\mathcal{M}_\mathcal{H}$ optimum partitioning as w_{OPT}. It holds that $w_{OPT} \geq (c_i(2n) + c_o(m))/J$, as each incoming tuple is assigned to at least one region. Since c_i and c_o are superadditive, it follows that $w_{OPT} \geq c_i(2n/J) + c_o(m/J)$ and that $\sigma \leq w_{OPT}/2$. More precisely, as the bucket sizes are probabilistic, we can conclude that with high probability, these bounds are very close to the actual bounds. ■

For the next lemma, we will need to define input and output sample sizes.

Input sample size s_i. For each relation, we build an approximate equi-depth histogram [13]. Namely, for each relation, we take a random uniform sample of size s_i, sort the sampled tuples according to the join key, and then build an equi-depth histogram on them with $n_s < s_i$ buckets.

According to [13], for a given n_s, s_i needs to be at least $4n_s \ln(2n/\gamma)/e^2$, where e is the maximum error on the bucket size with probability of at least $1-\gamma$. This implies that a small sample of size $s_i = \Theta(n_s \log n)$ is sufficient for building approximate equi-depth histogram.

Output sample size s_o. In [11], the authors show that the sample size is not a function of the actual data size, and that it can be obtained from standard tables based on Kolmogorov's statistics [15]. For example, for an error on the region *output* within 5% and confidence of at least 99%, the standard tables only require that the sample size is at least 1063. On the other hand, the sample size should be a small integer multiple of the

[xxi] Monotonicity on the join output and monotonicity on the weight function are different and should not be confused.

[xxii] This typically holds in practice. We relax it in §A5.

number of scrutinized categories in the population. In our case, this number is the number of candidate $\mathcal{M}_\mathcal{S}$ cells (n_{sc}), as the non-candidate cells never produce an output tuple. Thus, the output sample size is $s_o \geq \max(1063, n_{sc})$ for the specified error margin and confidence interval. Consequently, we need an output sample of size $s_o = \Theta(n_{sc})$.

To determine n_{sc}, we define a low-selectivity join precisely. As §II-A states, the content-insensitive (CI) scheme achieves (almost) perfect load balancing for *output*. Thus, if the output size $m > \rho_B n$ ($\rho_B \gg 1$ is a constant), CI works very well. Hence, it pays off to use a content-sensitive scheme only if $m \leq \rho_B n$ [xxiii] , that is, if $m = \mathcal{O}(n)$. This condition defines a *low-selectivity* join. We require a similar condition to hold between the input and output of the sample matrix[xxiv]:

$$n_{sc} = \mathcal{O}(n_s) \tag{1}$$

Thus, we need a small output sample: $s_o = \Theta(n_s) = \Theta(\sqrt{nJ})$.

We next prove that, given this $n_s = \sqrt{2nJ}$ from Lemma 3.1, the sampling stage time complexity is low.

Lemma 3.2 [Extended version]: The total sample size collected for building $\mathcal{M}_\mathcal{S}$ is $\Theta(n_s \log n)$. The sampling stage running time is $\mathcal{O}(n_s \log n_s)$. For $n_s = \sqrt{2nJ}$ and $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$ [xxv], the sample size and the time complexity are both $\mathcal{O}(n/J)$.

Proof: From $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$, it follows that

$$\log n = \mathcal{O}(\sqrt{n/J^3}) \tag{2}$$

As the input sample size is $s_i = \Theta(n_s \log n)$, and the output sample size is $s_o = \Theta(n_s)$, the total sample size is dominated by the input. By substituting $\log n$ from Equation 2, the total sample size is $s_i = \Theta(n_s \log n) = \Theta(\sqrt{nJ} \log n) = \mathcal{O}(n/J)$.

Let us discuss the time complexity for building $\mathcal{M}_\mathcal{S}$. To create approximate equi-depth histogram on input, we need to sort the input sample tuples. We do it on the sites providing the samples, incurring $\mathcal{O}(\log^2 s_i) = \mathcal{O}(\log^2(n/J))$ time.

For each sample output tuple (s_o of them), we use binary search to find a $\mathcal{M}_\mathcal{S}$ cell to increment. Thus, processing the output takes $\mathcal{O}(s_o \log n_s) = \mathcal{O}(n_s \log n_s)$ time. As $n_s \leq n$, it follows that $\log n_s \leq \log n$, and from Equation 2, $\log n_s = \mathcal{O}(\sqrt{n/J^3})$. Hence, $\mathcal{O}(n_s \log n_s) = \mathcal{O}(n/J)$. Thus, the total time complexity for building $\mathcal{M}_\mathcal{S}$ is bounded by $\mathcal{O}(n/J)$. ■

2) Coarsening: The coarsening algorithm [16] is the RTILE (rectangle tiling) problem with grid ($n_c \times n_c$) partitioning and the MAX-WEIGHT-ID metric. This is an approximation algorithm with an approximation ratio of 2 [16]. That is, given $\mathcal{M}_\mathcal{S}$ and n_c (the size of $\mathcal{M}_\mathcal{C}$), where the maximum $\mathcal{M}_\mathcal{C}$ cell weight of the optimum grid partitioning is ϕ_0, the algorithm returns an $\mathcal{M}_\mathcal{C}$ with maximum cell weight $\phi \leq 2\phi_0$.

For sparse matrices, if range trees are used for computing the prefix sum, the coarsening algorithm [16] runs in

$$\mathcal{O}(n_{sc} \log n_{sc} + (n_s + n_c^2 \log n_{sc}) \cdot n_c \log n_s) \tag{3}$$

time, where n_s is the $\mathcal{M}_\mathcal{S}$ size, and n_{sc} is the number of candidates in $\mathcal{M}_\mathcal{S}$.

MonotonicCoarsening. Monotonicity (consecutiveness of candidate cells) allows us to visit all the n_{cc} candidate cells in $\mathcal{O}(n_{cc})$ time. Along the lines of Equation 1, we assume $n_{cc} = \mathcal{O}(n_c)$. Consequently, the coarsening algorithm for monotonic joins performs only $\mathcal{O}(n_c)$, rather than n_c^2 weight computations per iteration. The complexity from Equation 3 then becomes:

$$\mathcal{O}(n_{sc} \log n_{sc} + (n_s + n_c \log n_{sc}) \cdot n_c \log n_s) \tag{4}$$

Lemma 3.3: The running time of the coarsening algorithm is $\mathcal{O}((n_s + n_c^2 \log n_s) \cdot n_c \log n_s)$. For $n_c = 2J$ and $J = \mathcal{O}(\sqrt[3]{n/\log^2 n_s})$ [xxvi], the time complexity becomes $\mathcal{O}(n)$.

Proof: Equation 3 shows the total running time for building the coarsened matrix. By substituting $n_{sc} = \mathcal{O}(n_s)$ (Equation 1), $\log n_s$ from $J = \mathcal{O}(\sqrt[3]{n/\log^2 n_s})$, $n_c = \Theta(J)$ and $n_s = \Theta(\sqrt{nJ})$ into Equation 3, it follows that the complexity is $\mathcal{O}(n)$. ■

3) Regionalization: Regionalization is an RTILE problem with arbitrary partitioning [17] and the MAX-WEIGHT-ID metric. There exist algorithms for this problem in the restricted, output-only case, e.g. [41]. However, they are not applicable for the general case, which entails support for: *a*) monotonic metrics (including the weight function) and *b*) segments which may or may not be covered by a region (0-cells). By design, these algorithms generate prolate regions and thus incur excessive *input* costs. Hence, they can be arbitrarily worse in weight than the optimum. The best algorithm which works for the general case is Binary Space Partition (BSP) [10], [17].

BSP. *Binary Space Partition* [10] is a dynamic programming algorithm which creates an optimum hierarchical partitioning, within a factor of 2 from an optimum arbitrary partitioning. We use the MAX-WEIGHT-ID metric.

Next, we prove lemmas about from §III-C.

Lemma 3.4: A rectangle is defined by the upper left and the lower right corner. For monotonic joins, each defining corner of a minimal candidate rectangle is a candidate cell, yielding $\mathcal{O}(n_c^2)$ minimal candidate rectangles in total.

Proof: We prove the lemma using contradiction by assuming that a defining corner of a minimal candidate region is not a candidate cell. Let us consider the position of the upper left corner. If it is before the first candidate cell in the row of $\mathcal{M}_\mathcal{C}$, the left boundary of the rectangle is empty (see rectangles r_1 and r_{min1} in Figure 2c). Thus, the rectangle is not a minimal candidate. If the position of the upper left corner is after the last candidate cell in the row, the upper boundary of the rectangle is empty (see rectangles r_2 and r_{min2} in Figure 2c). Again, the rectangle is not a minimal candidate. Consequently, the upper left corner must be a candidate cell. The proof for the lower right corner is symmetric. Thus, both defining corners of a minimal candidate rectangle are candidate cells.

[xxiii] In our setup, our scheme works well even if m is two orders of magnitude bigger than n (see §VI-C). Thus, we cover a wide range of joins in practice.

[xxiv] If any of these assumptions do not hold, we fall back to the content-insensitive operator (see §VI-E for details). However, we experimentally show that the assumptions hold for many interesting joins.

[xxv] See footnote vii.

[xxvi] See footnote vii.

Consequently, there are $n_{c_c}^2$ minimal candidate rectangles, where n_{cc} is the number of candidate cells in $\mathcal{M}_\mathcal{C}$. Along the lines of Equation 1, we assume $n_{cc} = \mathcal{O}(n_c)$. Thus, there are $\mathcal{O}(n_c^2)$ minimal candidate rectangles in total. ■

Lemma 3.5: The regionalization stage based on MONOTONICBSP runs in $\mathcal{O}(n_c^3 \log n_c \log n)$ time. For $n_c = 2J$ and $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$, the stage takes $\mathcal{O}(n)$ time.

Proof: Generating r_N minimal candidate rectangles and sorting them takes $\mathcal{O}(r_N \log r_N)$ time. Then, for each rectangle (there are r_N of them), we: *a*) compute its weight which takes $\mathcal{O}(1)$ time with $\mathcal{O}(n_c^2)$ prefix sum precomputation (line 16) and *b*) for each splitter line within a rectangle, $\mathcal{O}(n_c)$ of them, for both subrectangles, find the corresponding minimal candidate rectangle (using binary search it takes $\mathcal{O}(\log n_c)$ time) (lines 19-23). Step *b* yields $\mathcal{O}(n_c \log n_c)$ time per rectangle. Overall, this requires a total time of $\mathcal{O}(n_c^2 + r_N(\log r_N + n_c \log n_c))$.

From Lemma 3.4, we know that $r_N = \mathcal{O}(n_c^2)$. Thus, MONOTONICBSP runs in $\mathcal{O}(n_c^3 \log n_c)$ time. Due to transformation from DRTILE to RTILE, the regionalization stage takes $\mathcal{O}(n_c^3 \log n_c \log n)$ time. Given $n_c = \Theta(J)$, $\log J \leq \log n$ and $J = \mathcal{O}(\sqrt[3]{n/\log^2 n})$, it follows that the stage takes $\mathcal{O}(n)$ time. ■

4) Putting it all together:

Theorem 3.1 [Extended version]: The histogram algorithm runs in $\mathcal{O}(n)$ local time and it requires a total of $\mathcal{O}(n/J)$ sample tuples.

Proof: Lemmas 3.2, 3.3 and 3.5 directly imply Theorem 3.1. ■

We next discuss why this cost is affordable. As a parallel join takes $\Omega((n+m)/J)$ communication time (m is the join output size), the histogram algorithm can afford $\mathcal{O}(n/J)$ time for collecting sample tuples and $\mathcal{O}(n)$ local processing time (which is much cheaper than the communication time).

5) Generalization and Discussion: We next relax some assumptions and outline how we address them to preserve all the guarantees.

Heterogeneous clusters. In heterogeneous clusters, we assign work to machines proportionally to their capacity. To do so, we set the number of regions (J) in the histogram algorithm higher than the number of machines.

A small number of output tuples. We relax the assumption $m \geq n$ from Lemma 3.1. If $m < n$, a sample matrix $\mathcal{M}_\mathcal{S}$ cell frequency can surpass m/J, breaking the Lemma bounds. We distinguish two cases. (i) If $m = \Theta(n) = cn$, where $c < 1$ is a constant, we increase n_s to preserve the bounds. More precisely, it must hold that $(n/n_s)^2 \leq m/2J$, that is, $(n/n_s)^2 \leq cn/2J$. It follows that $n_s \geq \sqrt{2nJ/c}$. Thus, n_s grows only by a constant factor of $1/\sqrt{c}$. (ii) If $c \ll 1$ ($m \ll n$), to avoid a huge increase in n_s, and thus the histogram algorithm complexity, we adjust $\mathcal{M}_\mathcal{S}$ such that each cell weight is below the required threshold ($w_{OPT}/2$, where w_{OPT} is the maximum region weight of the $\mathcal{M}_\mathcal{H}$ optimum partitioning). Namely, we divide only the row and/or column of the overweighted cell(s). Then, we reassign the affected output sample tuples to the new $\mathcal{M}_\mathcal{S}$ cells.

Reducing the sample matrix size n_s is an important optimization, as it decreases the histogram algorithm running time. Lemma 3.1 decides on n_s using a conservative assumption that $\rho_B \geq 1$ in $m = \rho_B n$, that is, $m \geq n$. (We covered the case when $m < n$ earlier in this section.) Using the actual value of $\rho_B \geq 1$ decreases n_s from $\sqrt{2nJ}$ to $\sqrt{2nJ/\rho_B}$, without loosing any guarantees. We know m and thus ρ_B from sampling the output tuples (see §IV-A). This optimization requires rebuilding the sample matrix once m is known (input and output samples are collected as before). Reducing n_s is very useful when ρ_B is sufficiently bigger than 1 and when input relations are very large. We use it for B_{CB}.

Parameters. The output sample size is $s_o = \mathcal{O}(n_{sc})$. In our experiments we set $s_o = 2n_{sc}$. We compute n_{cs} by counting the candidate $\mathcal{M}_\mathcal{S}$ cells right after collecting a sample of input tuples.

B. Joins

The joins are defined as follows:

B_{ICD}

```
SELECT *
FROM  ORDERS O1, ORDERS O2
WHERE ABS(O1.orderkey - 10 * O2.custkey) <= 2
```

$B_{CB-\beta}$

```
SELECT *
FROM  R1, R2
WHERE ABS(R1.key - R2.key) <= β
```

BE_{OCD}

```
SELECT *
FROM  ORDERS O1, ORDERS O2
WHERE O1.custkey = O2.custkey
AND ABS(O1.ship-priority - O2.ship-priority) <= 2
AND O1.order-priority = "4-NOT SPECIFIED"
AND O2.order-priority = "1-URGENT"
AND O1.totalprice BETWEEN γ AND 360000
AND O2.totalprice BETWEEN γ AND 360000
```

For B_{CB}, we experiment with different widths of the band β: 1, 2, 3, 4, 8 and 16. Scaling out BE_{OCD} using the same γ leads to highly disturbed $output/input$ ratio ρ_{oi}. As the relative performance of different operators highly depends on ρ_{oi}, we set γ such that ρ_{oi} remains on the same order of magnitude. Namely, we set γ to 120.000, 140.000 and 160.000 for the scale factor of 80, 160 and 320, respectively.