Submodularity of Distributed Join Computation

Rundong Li
Northeastern University, Boston, USA
rundong@ccs.neu.edu

Mirek Riedewald
Northeastern University, Boston, USA
m.riedewald@northeastern.edu

Xinyan Deng
Northeastern University, Boston, USA
reena@ccs.neu.edu

ABSTRACT

We study distributed equi-join computation in the presence of join-attribute skew, which causes load imbalance. Skew can be addressed by more fine-grained partitioning, at the cost of input duplication. For random load assignment, e.g., using a hash function, fine-grained partitioning creates a tradeoff between load expectation and variance. We show that minimizing load variance subject to a constraint on expectation is a monotone submodular maximization problem with Knapsack constraints, hence admitting provably near-optimal greedy solutions. In contrast to previous work on formal optimality guarantees, we can prove this result also for self-joins and more general load functions defined as weighted sum of input and output. We further demonstrate through experiments that this theoretical result leads to an effective algorithm for the problem of minimizing running time, even when load is assigned deterministically.

CCS CONCEPTS

• **Information systems → Parallel and distributed DBMSs**; • **Theory of computation → Distributed algorithms**; • **Computing methodologies → Distributed algorithms**; *MapReduce algorithms*;

KEYWORDS

Distributed join; load balancing; load variance minimization

ACM Reference Format:
Rundong Li, Mirek Riedewald, and Xinyan Deng. 2018. Submodularity of Distributed Join Computation. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD/PODS '18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3183728

1 INTRODUCTION

We consider equi-joins in distributed environments, in particular clusters of physical machines or virtual machines in the cloud. While there is renewed interest in multiway joins, this work focuses on binary joins, including *self*-joins, which are the "bread and butter" join operators in distributed computation frameworks such as MapReduce and Spark, and in relational databases. (Queries joining more than two relations can be implemented through multiple binary joins.) We consider the most general problem scenario where (1) the join may or may not be a foreign key join, (2) load is defined as the weighted sum of input and output, and (3) the join attribute(s) in the given input might show any degree of skew.

The standard approach for joining two large data sets S and T is the classic algorithm relying on hash-based partitioning. It balances load well for low to moderate degree of join-attribute skew, but achieves poor speedup when high skew causes some workers to receive an excessive amount of load. This type of inherent skew can be addressed by a more fine-grained input partitioning. In particular, *heavy hitters*, i.e., join attribute values that occur with high frequency, can be split into smaller sub-groups—at the cost of input duplication. Consider the simple example of a two-worker cluster and inputs $S = S_a$ and $T = T_a$ where all tuples have the same join attribute value a. (Hence the equi-join here computes Cartesian product $S_a \times T_a$.) Hash partitioning assigns the entire input to one of the two workers, leaving the other idle. The resulting *max load* in the system, i.e., the load assigned to the worker receiving most load, then is $|S_a| + |T_a|$ input tuples and $|S_a||T_a|$ output tuples. Alternatively, one could split S_a in half and assign a partition to each worker. To still produce all output tuples, T_a has to be assigned to both workers. As a result, max load across workers improves to $|S_a|/2 + |T_a|$ input and $|S_a||T_a|/2$ output tuples.

The simple example highlights a tradeoff between total load and load balance. Partitioning S_a into two sub-groups increased total input by $|T_a|$, resulting in higher load overall. On the other hand, load could be distributed more evenly over the two workers, reducing the max load assigned to any worker. This raises several interesting questions: Which of the inputs should be partitioned? How much should it be partitioned? And how are these decisions affected by other join groups?

In general, our goal is to determine the partitioning of heavy-hitter join groups that minimizes the max load assigned to any worker. For random load assignment, max load is determined by load expectation and variance. In particular, given two partitionings with the same load expectation, we prefer the one with lower variance, because it has a higher probability of distributing the load evenly. Hence we study the problem of *minimizing load variance subject to a threshold on load expectation.*

Our work is the first to provide formal optimality guarantees that (1) do not ignore constant factors and (2) extend to a more general load function accounting for both input and output. We identify precise conditions that guarantee it to be a monotone submodular optimization problem with Knapsack constraints, admitting a greedy solution with strong optimality guarantees. Intuitively, this novel algorithm aims to maximize variance reduction per input duplicate generated by a sub-group partitioning.

While these theoretical results are the main objective, they also lead to an effective practical algorithm for *deterministic* load assignment. We hypothesize that optimizing for lower variance makes the partitions "easier" to distribute evenly, striking the right balance

SIGMOD/PODS '18, June 10–15, 2018, Houston, TX, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183728

Variable	Meaning
w	number of worker nodes in the cluster
S, T	input relations
A	join attribute and set of its possible values
H	$H \subseteq A$ is the set of heavy hitter candidates
S_a, T_a	$S_a = \{s \in S : s.A = a\}$, $T_a = \{t \in T : t.A = a\}$
p_h	for rectangular partitioning: number of partitions for input S_h, $h \in H$; for triangular partitioning: number of rows and columns of upper triangular matrix
q_h	only for rectangular partitioning: number of partitions for input T_h, $h \in H$
$\mathbb{E}[L]$	expectation of random variable L
$\mathbb{V}[L]$	variance of random variable L
α, γ	load weight factors
Ω	finite set of elements for representing sub-group partitionings
$\mathcal{M}$	maps a set $X \subseteq \Omega$ to a rectangular sub-group partitioning $[(p_h, q_h)]_{h \in H}$ or triangular sub-group partitioning $[p_h]_{h \in H}$
$\mathcal{R}$	cost model to estimate running time of distributed join computation

Table 1: Important notation

between input duplication and load balance. We demonstrate this empirically in an extensive experimental study; establishment of a formal relationship is left for future work. This work makes the following main contributions:

- We precisely quantify load expectation and variance for state-of-the-art sub-group partitioning strategies under random load assignment. This analysis uses the most general load model from the literature and also applies to self-joins and equi-joins that are not foreign-key joins.
- We establish a relationship between load expectation, variance, and max load through Chebyshev's inequality, leading to the optimization problem of minimizing load variance subject to a threshold on its expectation.
- For this optimization problem, we derive precise conditions that guarantee it to be a monotone submodular maximization problem with Knapsack constraints. Based on this result, we propose novel near-optimal greedy solutions.
- We demonstrate experimentally how the theoretical results can be leveraged to reduce running time compared to the state of the art for distributed join computation, even when load is assigned deterministically.

2 FOUNDATIONS

Table 1 provides an overview of important notation.

Join problem: We consider the equi-join between two data sets $S = \{s_1, s_2, \ldots, s_{|S|}\}$ and $T = \{t_1, t_2, \ldots, t_{|T|}\}$, where $|X|$ denotes the cardinality of set X. This includes the special case of a *self*-join when $S = T$. Without loss of generality, let the schemas of S and T both contain an attribute A on which both are joined. Equi-joins on multiple attributes can be converted to an equivalent join on a single attribute. To avoid clutter, we follow common practice and slightly abuse notation by using A also to denote the set of possible values of this attribute—context will make the meaning clear.

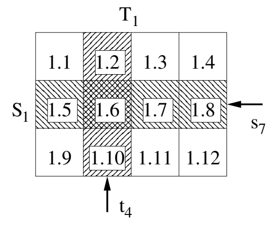

Figure 1: Join group 1 is partitioned into $p_1 = 3$ by $q_1 = 4$ sub-groups. Each $s \in S_1$ has to be assigned to a row, requiring q_a copies of s. Similarly, each $t \in T_1$ is assigned to a column. The choice of row and column does not matter for correctness. For even load distribution over the sub-groups, rows and columns are chosen uniformly at random.

Our goal is to compute the equi-join $S \bowtie T = \{(s_i, t_j) \in S \times T : s_i.A = t_j.A\}$ using w worker nodes in the system. Let $S_a = \{s \in S : s.A = a\}$ and $T_a = \{t \in T : t.A = a\}$ be the subsets of tuples from S and T, respectively, with join attribute value a. We will refer to $S_a \cup T_a$ as the *group* for join attribute value a. Then the equi-join can be expressed as

$$S \bowtie T = \bigcup_{a \in A} S_a \times T_a,$$

i.e., the union of Cartesian products for each group.

Like all previous work on joins with skewed input, we assume that simple count statistics for "large" join groups are given. In contrast to previous work, the set of heavy hitters does not have to given. Our approach can work with any set $H \subseteq A$ of heavy hitter *candidates*, determining automatically which ones (not) to partition.

Sub-group partitioning: For each $a \in A$, (p_a, q_a) defines a uniform sub-group partitioning of join group a. There are $p_a q_a$ sub-groups, each receiving $|S_a|/p_a$ tuples from S_a and $|T_a|/q_a$ tuples from T_a. As a consequence, each sub-group produces $\frac{|S_a||T_a|}{p_a q_a}$ output tuples. Figure 1 shows an example for a hypothetical group 1 with $p_1 = 3$ and $q_1 = 4$. Records such as $s_7 \in S_1$ and $t_4 \in T_1$ are randomly assigned a row number between 1 and p_a or column number between 1 and q_a, respectively. This implies that the input tuple is assigned to all sub-groups in the corresponding row or column. In the example, s_7 was assigned to 1.5, 1.6, 1.7, and 1.8. Similarly, t_4 was assigned to sub-groups 1.2, 1.6, and 1.10. Notice that exactly one sub-group (1.6 in the example) receives both s_7 and t_4. Hence result (s_7, t_4) will be output *exactly* once—by the worker processing sub-group 1.6.

Previous work showed that this style of *rectangular* partitioning achieves asymptotically near-optimal load assignment for Cartesian product [27] and equi-join [2]. Those results apply to the general case. For the special case of *natural* self-joins, e.g., `SELECT * FROM S AS S1, S AS S2 WHERE S1.A = S2.A`, *triangular* partitioning, also provably near-optimal, was introduced to reduce input duplication [6]. For self-joins on different attributes, e.g., `SELECT * FROM S AS S1, S AS S2 WHERE S1.A = S2.B`, triangular partitioning

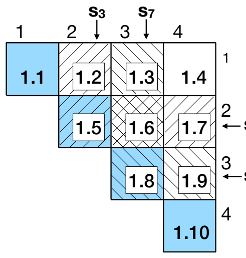

Figure 2: Join group 1, for $p_1 = 4$, is partitioned into $(p_1 + 1)p_1/2 = 10$ sub-groups. Each $s \in S_1$ has to be assigned to a row and column with the same index, requiring p_1 copies of s. The choice of row/column does not matter for correctness. For even load distribution over the sub-groups, rows and columns are chosen uniformly at random.

cannot be applied. In the rest of the paper, we will use *self-join* as a shorthand for natural self-join.

Instead of a p_a-by-q_a matrix, it creates an upper triangular matrix with p_a rows and columns as illustrated in Figure 2. An input tuple is randomly assigned to a number between 1 and p_a, implying its assignment to all sub-groups in the corresponding row and column. In the example, s_3 was assigned to row and column 2, i.e., sub-groups 1.2, 1.5, 1.6, and 1.7. Tuple s_7 was assigned to row/column 3, corresponding to sub-groups 1.3, 1.6, 1.8, and 1.9. Similar to the rectangular partitioning, there is exactly one sub-group (1.6 in the example) that receives both inputs. It—and only it—is the one to produce output pairs (s_3, s_7) and (s_7, s_3). Note that the join has to output both results, while the duplication problem for which the partitioning was introduced, only required one of the two. (This slightly widens the gap between lower and upper bound for output load by a small constant factor between 1 and 2 compared to the results shown in [6].)

In general, each sub-group on the diagonal is assigned $|S_a|/p_a$ input tuples, outputting all $|S_a|^2/p_a^2$ pairs. Off-diagonal sub-groups each receive $2|S_a|/p_a$ input tuples—half "vertically" and half "horizontally" (see shading in Figure 2). They each produce $2|S_a|^2/p_a^2$ output tuples—all pairs of "vertical" and "horizontal" inputs in both orders, (horizontal, vertical) and (vertical, horizontal).

3 LOAD EXPECTATION AND VARIANCE

In previous work, load was defined as the amount of input, output, or a linear combination of both. We use the most general of these models and define load as the weighted sum $\gamma I + \alpha O$, for $\gamma, \alpha \geq 0$, where I and O denote input and output cardinality, respectively. This general load definition is needed, because in practice per-input and per-output tuple cost can vary significantly, e.g., depending if the join result is kept in memory (for further processing) or written to files.

Let l_i denote the load induced by (sub) group i. To analyze the total load assigned to a worker, we define random variable L_i whose value is l_i if the sub-group is assigned to that worker, and zero otherwise. Note that when assigning (sub) groups randomly, then L_i is independent of all L_j for any other (sub) group j. Expected load L is defined as

$$\mathbb{E}[L] = \mathbb{E}[\sum_i L_i] = \sum_i \mathbb{E}[L_i] = \frac{1}{w}\sum_i l_i \tag{1}$$

Variance is defined as $\mathbb{V}[L] = \mathbb{E}[L^2] - \mathbb{E}^2[L]$. From Eq. 1 follows

$$\mathbb{E}^2[L] = (\frac{1}{w}\sum_i l_i)^2 = \frac{1}{w^2}\sum_i\sum_j l_i l_j. \tag{2}$$

For $\mathbb{E}[L^2]$, we obtain

$$\begin{aligned}\mathbb{E}[L^2] &= \mathbb{E}[(\sum_i L_i)^2] = \mathbb{E}[\sum_i\sum_j L_i L_j] = \sum_i\sum_j \mathbb{E}[L_i L_j] \\ &= \sum_i\sum_{j\neq i}\mathbb{E}[L_i L_j] + \sum_i \mathbb{E}[L_i^2] = \frac{1}{w^2}\sum_i\sum_{j\neq i} l_i l_j + \frac{1}{w}\sum_i l_i^2.\end{aligned} \tag{3}$$

This derivation uses that L_i^2 takes on value l_i^2 with probability $1/w$, and zero otherwise. From Eq. 2 and 3 follows

$$\mathbb{V}[L] = \frac{1}{w}\sum_i l_i^2 - \frac{1}{w^2}\sum_i l_i^2 = \frac{w-1}{w^2}\sum_i l_i^2 \tag{4}$$

The actual load amounts l_i depend on the sub-group partitioning. We next analyze rectangular and triangular partitioning separately.

3.1 Rectangular Partitioning

As discussed in Section 2, each heavy hitter group $h \in H$ is partitioned into p_h-by-q_h equal-sized sub-groups, where $p_h \geq 1, q_h \geq 1$. Setting $p_h = q_h = 1$ implies no sub-group partitioning for this group. Sub-group partitioning creates $p_h q_h$ subgroups, each with $|S_h|/p_h + |T_h|/q_h$ input and $|S_h||T_h|/(p_h q_h)$ output tuples. Together with Eq. 1 and Eq. 4, this implies for the load induced by the heavy-hitter (sub) groups on a worker

$$\begin{aligned}\mathbb{E}[L] &= \frac{1}{w}\sum_{h\in H} p_h q_h \left(\gamma\frac{|S_h|}{p_h} + \gamma\frac{|T_h|}{q_h} + \alpha\frac{|S_h||T_h|}{p_h q_h}\right) \\ &= \frac{1}{w}\sum_{h\in H}(\gamma q_h|S_h| + \gamma p_h|T_h| + \alpha|S_h||T_h|).\end{aligned} \tag{5}$$

$$\begin{aligned}\mathbb{V}[L] &= \frac{w-1}{w^2}\sum_{h\in H} p_h q_h \left(\gamma\frac{|S_h|}{p_h} + \gamma\frac{|T_h|}{q_h} + \alpha\frac{|S_h||T_h|}{p_h q_h}\right)^2 \\ &= \frac{w-1}{w^2}\sum_{h\in H}\left(\gamma^2\frac{q_h}{p_h}|S_h|^2 + \gamma^2\frac{p_h}{q_h}|T_h|^2 + \frac{\alpha^2}{p_h q_h}|S_h|^2|T_h|^2\right. \\ &\qquad \left. + 2\gamma^2|S_h||T_h| + \frac{2\alpha\gamma}{p_h}|S_h|^2|T_h| + \frac{2\alpha\gamma}{q_h}|S_h||T_h|^2\right).\end{aligned} \tag{6}$$

3.2 Triangular Partitioning

As discussed in Section 2, each heavy hitter group $h \in H$ is partitioned into an upper triangular matrix of p_h rows and columns. Setting $p_h = 1$ implies no sub-group partitioning for this group. We need to distinguish between sub-groups on and off the diagonal. There are p_h sub-groups on the diagonal, each with $|S_h|/p_h$ input and $|S_h|^2/p_h^2$ output. The remaining $p_h(p_h - 1)/2$ sub-groups have $2|S_h|/p_h$ input and $2|S_h|^2/p_h^2$ output. Together with Eq. 1 and Eq. 4, this implies for the load induced by the heavy-hitter (sub) groups on a worker

$$\mathbb{E}[L] = \frac{1}{w}\sum_{h\in H}\left(p_h\left(\gamma\frac{|S_h|}{p_h}+\alpha\frac{|S_h|^2}{p_h^2}\right)\right.$$
$$\left.+\frac{p_h(p_h-1)}{2}\left(\gamma\frac{2|S_h|}{p_h}+\alpha\frac{2|S_h|^2}{p_h^2}\right)\right)$$
$$=\frac{1}{w}\sum_{h\in H}\left(\gamma p_h|S_h|+\alpha|S_h|^2\right) \tag{7}$$

$$\mathbb{V}[L] = \frac{w-1}{w^2}\sum_{h\in H}\left(p_h\left(\gamma\frac{|S_h|}{p_h}+\alpha\frac{|S_h|^2}{p_h^2}\right)^2\right.$$
$$\left.+\frac{p_h(p_h-1)}{2}\left(\gamma\frac{2|S_h|}{p_h}+\alpha\frac{2|S_h|^2}{p_h^2}\right)^2\right)$$
$$=\frac{w-1}{w^2}\sum_{h\in H}\left(2-\frac{1}{p_h}\right)\left(\gamma|S_h|+\frac{\alpha}{p_h}|S_h|^2\right)^2. \tag{8}$$

4 LOAD OPTIMIZATION

Worker load L is a random variable and the one-sided version of Chebyshev's inequality states that for any random variable X and constant $k > 0$, $\Pr(X \geq \mathbb{E}[X]+k) \leq \frac{\mathbb{V}[X]}{\mathbb{V}[X]+k^2}$. Setting $k = 10\sqrt{\mathbb{V}[X]}$ and applying this inequality to load L, we obtain

$$\Pr(L \geq \mathbb{E}[L] + 10\sqrt{\mathbb{V}[L]}) \leq 1/(1+10^2) < 0.01,$$

i.e., the probability of a worker's load to exceed load expectation by more than 10 standard deviations is less than 1%. Given two load configurations with the same expectation, we therefore prefer the one with lower standard deviation (and hence variance). This leads to the following optimization problem of *minimizing load variance subject to a threshold on load expectation.* Maximizing negative variance is equivalent to minimizing variance.

Problem 1. *Maximize* $-\mathbb{V}[L]$ *subject to* $\mathbb{E}[L] \leq \theta$.

We next prove structural properties of this optimization problem that will enable fast greedy heuristics with strong approximation guarantees. In particular, we show this problem to be a monotone submodular maximization problem with Knapsack constraints.

4.1 Monotonicity Properties

For monotonicity, we prove strong results for both rectangular and triangular sub-group partitioning. Due to their popularity in database queries, the special case of foreign-key joins is listed explicitly. Note that in the input where the join attribute is the key, cardinality of the group is equal to 1. All proofs are in the appendix. They mathematically derive the change in variance as the sub-group partitioning becomes more fine-grained.

Theorem 4.1. *Negative variance* $-\mathbb{V}[L]$ *for rectangular partitioning* $[(p_h, q_h)]_{h\in H}$ *is monotonically increasing in* p_h *and* q_h, $h \in H$, *if* $\alpha \geq \gamma(p_h+1)/|S_h|$ *and* $\alpha \geq \gamma(q_h+1)/|T_h|$.

Theorem 4.2. *For the special case of a foreign key join, i.e.,* $|T_h| = q_h = 1$ *for all* $h \in H$, *negative variance* $-\mathbb{V}[L]$ *for rectangular partitioning* $[(p_h, q_h)]_{h\in H}$ *is monotonically increasing in* p_h.

Theorem 4.3. *Negative variance* $-\mathbb{V}[L]$ *for triangular partitioning* $[p_h]_{h\in H}$ *is monotonically increasing in* p_h, $h \in H$, *if* $\alpha \geq \gamma p_h/|S_h|$.

4.2 Submodularity Properties

Submodularity is a property of some discrete functions defined over sets. Intuitively, a submodular function shows *diminishing returns*, i.e., adding a new element to set X results in greater increase of the function value than adding it to a superset of X.

Definition 4.4. Let Ω be a finite set and $f : 2^\Omega \rightarrow \mathbb{R}$ be a function that maps any subset of Ω to a real number. Function f is *submodular* if for every $X \subseteq Y \subseteq \Omega$ and every $x \in \Omega - Y$, $f(X \cup \{x\}) - f(X) \geq f(Y \cup \{x\}) - f(Y)$.

To be able to apply the concept of submodularity to our load optimization problem, its objective function $-\mathbb{V}[L]$ has to be converted into a function over sets. We propose the following construction. Recall that $-\mathbb{V}[L]$ is defined over input tuples of the type $[(p_h, q_h)]_{h\in H}$ and $[p_h]_{h\in H}$ for rectangular and triangular partitioning, respectively. To convert such a partitioning into a set, we represent more fine-grained partitionings with larger sets. In particular, any number n can be represented as a set of n elements. However, we also need to distinguish p_h versus q_h, and between the different $h \in H$. We achieve this by defining each element as a triple (h, i, j), where $h \in H$ identifies to which join group the element belongs, i distinguishes between the inputs (p's versus q's; omitted for triangular partitioning), and j distinguishes between the elements for the same join group and input. Formally, define

$$\Omega = \bigcup_{h\in H, 1\leq j\leq |S_h|}\{(h,0,j)\} \quad \cup \bigcup_{h\in H, 1\leq j\leq |T_h|}\{(h,1,j)\}.$$

For example, assume there are two heavy hitter groups $h = 1$ and $h = 2$ with $|S_1| = 4$, $|S_2| = 2$, $|T_1| = 1$, and $|T_2| = 3$. Then $\Omega = \{(1,0,1),(1,0,2),(1,0,3),(1,0,4),(2,0,1),(2,0,2),(1,1,1),(2,1,1),(2,1,2),(2,1,3)\}$. Sub-group partitioning with $p_1 = 2$, $p_2 = 1$, $q_1 = 1$, and $q_2 = 2$ can then be expressed by set $\{(1,0,1),(1,0,2),(2,0,1),(1,1,1),(2,1,1),(2,1,2)\}$, but also $\{(1,0,2),(1,0,4),(2,0,2),(1,1,1),(2,1,1),(2,1,3)\}$ etc. In general, value p_h is encoded by a set of p_h elements of type $(h,0,*)$ from Ω; and analogously $(h,1,*)$ for q_h. Incrementing p_h by 1 then corresponds to adding another element of type $(h,0,*)$ to this set.

For the reverse direction, i.e., to map any subset of Ω to a sub-group partitioning, we define a mapping function $\mathcal{M}$ that is defined as follows: Given $X \subseteq \Omega$, for each $h \in H$, $\mathcal{M}$ sets p_h to the number of elements whose first field has value h and second field 0 (i.e., it counts all $(h,0,*)$) in X, and q_h to the number of elements whose first and second field are h and 1, respectively (i.e., it counts all $(h,1,*)$) in X.

For rectangular partitioning, we can now express objective function $-\mathbb{V}[L]$ in Problem 1 equivalently with set function f_1:

$$f_1(X) = -\sum_{h\in H} p_h q_h\left(\gamma\frac{|S_h|}{p_h}+\gamma\frac{|T_h|}{q_h}+\alpha\frac{|S_h||T_h|}{p_h q_h}\right)^2, \tag{9}$$

where $X \subseteq \Omega$ and $[(p_h, q_h)]_{h\in H} = \mathcal{M}(X)$. Recall that by construction of $\mathcal{M}$, adding an element $x \in \Omega - X$ to X has the effect that the corresponding p_h or q_h increases by 1, while all others remain unaffected. Hence f_1 is monotonically increasing if and only if the corresponding variance function is as well (see Section 4.1).

For triangular partitioning, we define a monotone set function f_2 analogously as

$$f_2(X) = -\sum_{h\in H}\left(2-\frac{1}{p_h}\right)\left(\gamma|S_h| + \frac{\alpha}{p_h}|S_h|^2\right)^2. \qquad (10)$$

We can now prove submodularity of f_1 and f_2 by analyzing their discrete derivatives. This yields the following strong results; all proofs are in the appendix.

Theorem 4.5. *Function f_1 for rectangular partitioning $[(p_h, q_h)]_{h\in H}$ is submodular if $\alpha \geq \sqrt{2}(p_h+1)\gamma/|S_h|$ and $\alpha \geq \sqrt{2}(q_h+1)\gamma/|T_h|$.*

Theorem 4.6. *For the special case of a foreign key join, i.e., $|T_h| = q_h = 1$ for all $h \in H$, function f_1 is submodular.*

Theorem 4.7. *Function f_2 for triangular partitioning $[p_h]_{h\in H}$ is submodular if $\alpha \geq \gamma p_h/|S_h|$.*

4.3 Knapsack Constraints

A Knapsack constraint on set $X \subseteq \Omega$ is a constraint of the type

$$\sum_{x\in X} c(x) \leq B,$$

where $c(x)$ is the non-negative integer "weight" of an element x and B is a non-negative threshold value [21]. This section shows that constraint $\mathbb{E}[L] \leq \theta$ of Problem 1 is such a Knapsack constraint.

Recall that Section 4.2 defined the elements in Ω to be triples of type (h, i, j), where h denotes the group and i the input (S versus T). For rectangular partitioning, set $c(h, 0, j) = |T_h|$ and $c(h, 1, j) = |S_h|$. This models the fact that increasing p_h to $p_h + 1$ requires one additional copy of T_h to be assigned to the corresponding sub-groups (and analogously for increasing q_h). For triangular partitioning, set $c(h, 0, j) = |S_h|$, because adding another row and column to the upper triangular matrix results in an extra copy of S_h.

For rectangular partitioning, based on Eq. 5, constraint $\mathbb{E}[L] \leq \theta$ is equivalent to

$$\sum_{h\in H}(q_h|S_h| + p_h|T_h|) \leq \left(w\theta - \sum_{h\in H}\alpha|S_h||T_h|\right)/\gamma.$$

It is easy to see that for any $X \subseteq \Omega$ that contains p_h elements of type $(h, 0, *)$ and q_h elements of type $(h, 1, *)$, the sum of the corresponding weights $c(x)$ equals $\sum_{h\in H}(q_h|S_h| + p_h|T_h|)$. Since w, θ, γ, and $\sum_{h\in H}\alpha|S_h||T_h|$ are independent of the sub-group partitioning, the right side $(w\theta - \sum_{h\in H}\alpha|S_h||T_h|)/\gamma$ is a constant as desired. The analysis for triangular partitioning is analogous.

4.4 Near-Optimal Greedy Heuristic

Submodularity can be exploited for efficiently and near-optimally solving Problem 1 using fast greedy heuristics. They find Pareto-optimal combinations of load expectation and variance.

For a non-negative monotone submodular function with Knapsack constraints, a simple greedy algorithm finds a solution that is guaranteed to be within factor $(1 - 1/e)/2$ of the optimal max value [21]. (Functions f_1 and f_2 can be made non-negative by adding a sufficiently large constant. This does not affect monotonicity or submodularity.) This algorithm idea applied to our problem results in novel join partitioning Alg. 2. It starts with the given join groups, i.e., $p_h = 1$ and $q_h = 1$ for all $h \in H$. It then greedily increases the p_h or q_h with the greatest ratio of variance reduction to input duplication increase, i.e., it uses a *cost-benefit* greedy partitioning approach. Sometimes this strategy can diverge significantly from the optimal solution. To guarantee the $(1 - 1/e)/2$ approximation ratio, it is sufficient to execute the greedy partitioning also for *uniform* weights and choose the better of the two solutions.

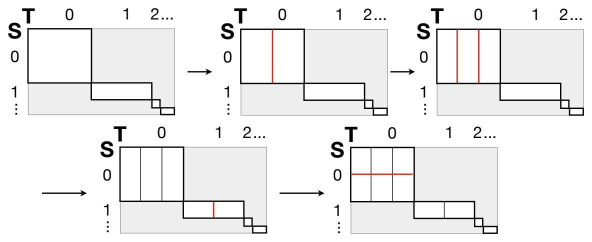

Figure 3: Example of GreedyPartition steps

Algorithm 1 : GreedyPartition

```
Input: heavy hitter candidate statistics $[|S_h|]_{h\in H}$, $[|T_h|]_{h\in H}$
Input: starting partitioning $[(p_h, q_h)]_{h\in H}$
Input: weight function weight
1: while $\sum_{h\in H}(q_h|S_h| + p_h|T_h|) \leq \theta$ do
2:     Increment the $p_h$ or $q_h$ that maximizes the ratio of benefit
       and cost
3:     /* Benefit = amount of variance reduction when incrementing
       the corresponding $p_h$ or $q_h$ */
4:     /* Cost = weight assigned by weight to the corresponding
       $p_h$ or $q_h$ */
5: end while
6: Undo the last increment operation /* It exceeded the threshold.
   */
7: return new partitioning $[(p_h, q_h)]_{h\in H}$
```

Figure 3 illustrates a sequence of GreedyPartition's steps. Even if all join groups are declared as potential "heavy hitters", it automatically focuses on splitting the largest ones first. Once their sub-groups are sufficiently small, it will start partitioning the medium ones, etc; possibly increasing partition number for large groups further in a later iteration.

Algorithm analysis: Alg. 1 has low computational complexity. In each iteration of the while-loop, it computes cost and benefit for all p_h and q_h. For cost, this involves a simple array lookup of the weight vector. For benefit, $\mathbb{V}[L]$ is re-computed. Since incrementing either p_h or q_h affects only 4 terms in Eq. 6 and 2 terms in Eq. 8 when incrementing p_h, re-computation time is constant as well. Hence the loop in Alg. 1 has computational complexity O($|H|$).

5 FROM LOAD TO RUNNING TIME

This section demonstrates how the above theoretical results, in particular Alg. 2, can be leveraged for designing a highly effective skew-resistant equi-join algorithm for practical use. End users in practice care about the time it takes to complete their job—the *running time* of the distributed execution. To optimize for running time, we follow a standard approach to tie load to running time.

Algorithm 2 : SimpleGreedy

```
1: $P_1$ = GreedyPartition($[|S_h|]_{h \in H}$, $[|T_h|]_{h \in H}$, $[(p_h = 1, q_h = 1)]_{h \in H}$, $(\mathtt{weight}(p_h) = |T_h|, \mathtt{weight}(q_h) = |S_h|)_{h \in H}$)
2: $P_2$ = GreedyPartition($[|S_h|]_{h \in H}$, $[|T_h|]_{h \in H}$, $[(p_h = 1, q_h = 1)]_{h \in H}$, $(\mathtt{weight}(p_h) = 1, \mathtt{weight}(q_h) = 1)_{h \in H}$)
3: return the partitioning $P_i$, $i \in \{1, 2\}$, that has lower variance $\mathbb{V}[L]$
```

First, we need to estimate how an individual worker's running time depends on its assigned input and output. This is a classic DBMS join cost estimation problem. Second, since data needs to be moved through the network to make input available to the appropriate workers, communication cost has to be taken into account. For this, we need to estimate network transfer time (plus time to deliver the data to the application running on the worker)—also a standard problem. We demonstrate this for the popular Reduce-side join in MapReduce, an implementation of the classic hash join. Note that the approach can be extended to implementation in other distributed systems, e.g., Spark or a distributed DBMS, by adjusting execution and data transfer cost estimation accordingly.

5.1 Join Algorithm

Reduce-side join does not make any assumptions about the initial location of the given inputs in the distributed file system. It performs a complete shuffle, assigning groups to workers (i.e., reduce tasks) using a hash function. Given a sub-group partitioning $[(p_h, q_h)]_{h \in H}$, it is straightforward to extend the algorithm by treating each *sub-group* as a separate join group. In summary, the algorithm consists of the following steps:

(1) In the map phase, each worker reads (often locally stored) input tuples, assigning them—and replicating if necessary—to the corresponding reduce tasks.
(2) The input tuples are shuffled across the network so that each worker (reduce task) receives its assigned (sub) groups.
(3) In the reduce phase, each worker processes the (sub) groups assigned to it and emits the output records.

Algorithms 3 and 4 show the MapReduce pseudo-code. The self-join implementation using the triangular partitioning is analogous.

5.2 Minimizing Running Time

Algorithm 1 finds the sub-group partitioning with near-minimal variance, subject to a constraint θ on load expectation. With increasing θ, load will be more evenly distributed across the workers (due to lower variance), but the total amount of load increases (due to higher expectation). Hence one has to determine the value of θ that strikes the best balance between the two, such that running time is minimized. In the join algorithm introduced in Section 5.1, sub-group partitioning affects the amount of data emitted by the mappers and then shuffled/sorted across the network, and the amount of input and output assigned to each reducer. Estimating data transfer time depending on data size and local join processing cost on a single worker are standard cost estimation problems. We use a

Algorithm 3 : Map

```
Input: partitioning $[(p_h, q_h)]_{h \in H}$ (from distributed file cache)
Input: input tuple $s_h \in S_h$ or $t_h \in T_h$
1: if input tuple is $s_h \in S_h$ then
2:     row = random(0, $p_h - 1$) /* Choose random integer from range $[0, p_h - 1]$ */
3:     /* Emit tuple copy to each sub-group in selected row */
4:     for subKey = (row * $q_h$) to (row * $q_h + q_h - 1$) do
5:         emit( ($h$, subKey), $s_h$ )
6:     end for
7: else
8:     col = random(0, $q_h - 1$) /* Choose random integer from range $[0, q_h - 1]$ */
9:     /* Emit tuple copy to each sub-group in selected column */
10:    for subKey = col to (($p_h - 1)q_h$ + col) step $q_h$ do
11:        emit( ($h$, subKey), $t_h$ )
12:    end for
13: end if
```

Algorithm 4 : Reduce

```
Input: key = ($h$, subKey); list of values of type $s_h \in S_h$, $t_h \in T_h$
1: Compute the cross-product between all $s_h \in S_h$ and $t_h \in T_h$ in the input value list; emit all these pairs
```

simple linear estimation model[1] defined as

$$\mathcal{R} = \beta_0 + \beta_1 I_t + \beta_2 I_m + \beta_3 O_m. \tag{11}$$

Here I_t denotes the total amount (in bytes) of reducer input (and hence mapper output) shuffled/sorted, and I_m and O_m are the amount (in bytes) of input and the cardinality of output, respectively, assigned to the worker receiving the highest load. We train a different such model for different ranges of output tuple sizes (in bytes), using a logarithmic scale: tuple size between 0 to 100 bytes, 101 to 1000 bytes, 1001 to 10,000 bytes etc.

The model parameters are obtained using linear regression from training runs, where we vary the total amount of data shuffled by the mappers in order to estimate β_0 and β_1. Parameters β_2 and β_3, which correspond to γ and α, respectively, in the load model, are estimated by executing joins for varying input and output sizes. Essentially, we estimate the data transfer speed of the network and the local join processing time at the individual worker machines. Hence only small training data is needed: we obtained good estimates from a few dozen training runs.

Algorithm 5, which we refer to as **GreedyPartition-R**, shows the corresponding version of GreedyPartition (Alg. 1) with running-time model $\mathcal{R}$. The concrete termination condition for the while-loop is discussed below. In line 5, to apply running time model $\mathcal{R}$, the values of I_t, I_m, and O_m need to be derived from sub-group partitioning $[(p_h, q_h)]_{h \in H}$. This is trivial for $I_t = \sum_{h \in H}(q_h|S_h|\sigma_S + p_h|T_h|\sigma_T)$, where σ_S and σ_T denote the size (in bytes) of an input tuple from S and T, respectively. However, I_m, and O_m depend on the assignment of (sub) groups to workers. Our algorithm can consider

[1]More detailed and accurate models could be used, but our experiments indicate that even this simple estimate is sufficient for capturing the main running time trends.

Algorithm 5 : GreedyPartition-R

```
Input: heavy hitter candidate statistics $[|S_h|]_{h\in H}$, $[|T_h|]_{h\in H}$
Input: starting partitioning $[(p_h, q_h)]_{h\in H}$
Input: weight function weight
1: while termination condition not met do
2:     Increment the $p_h$ or $q_h$ that maximizes the ratio of benefit and cost
3:     /* Benefit = amount of variance reduction when incrementing the corresponding $p_h$ or $q_h$ */
4:     /* Cost = weight assigned by weight to the corresponding $p_h$ or $q_h$ */
5:     Determine $I_t$, $I_m$, and $O_m$ for the new partitioning
6:     Evaluate running-time model $\mathcal{R}$ for the new $I_t$, $I_m$, $O_m$
7: end while
8: return partitioning $[(p_h, q_h)]_{h\in H}$ with lowest predicted running time
```

any load assignment function, including random and deterministic scheduling heuristics.

For random assignment of (sub) groups to workers, one typically uses hash functions, in particular *universal hashing*. A hash function maps each (sub) group key to a worker. In order to increase the probability of finding a well-balanced (sub) group assignment, this approach would use $k' > 1$ hash functions, selected randomly from the universal family.

For deterministic load assignment, note that minimizing $\beta_2 I_m + \beta_3 O_m$ corresponds to a classic minimum makespan scheduling problem that is known to be NP-hard. A simple greedy heuristic was shown to give a 4/3-approximation [13]. It assigns (sub) groups in decreasing order of their induced load, i.e., $\beta_2(\frac{|S_h|\sigma_S}{p_h} + \frac{|T_h|\sigma_T}{q_h}) + \beta_3\frac{|S_h||T_a|}{p_h q_h}$, one-by-one, always to the worker with the least load. We will refer to this algorithm as least-loaded decreasing (**LLD**). The running time of LLD is $k \log k$, where $k = \sum_{h\in H} p_h q_h$ denotes the number of heavy hitter (sub) groups.

While-loop termination condition. As the greedy algorithm increments the number of partitions for heavy-hitter groups, max load will initially decrease as big groups are broken up and load can be more evenly distributed. (This is not a monotonic behavior: max load might rise and fall repeatedly in the process.) At the same time, average load will monotonically increase, because the number of input duplicates increases. Figure 4 shows a typical behavior we observed in our experiments. The first few partitionings quickly reduce the gap between max and average load, indicating rapidly improving load balance. As this gap closes, the potential for running time reduction from better load distribution diminishes and the steadily increasing average load begins to have a dominating effect on running time. This causes running time to first drop, then reach a minimum, from which it steadily starts increasing again. To ensure reliable identification of this turning point, we propose the following while-loop termination condition: model-estimated running time dropped by less than 1% total over the last w iterations. In our experiments, even for heavily skewed data, the algorithm never exceeded 100 loop executions.

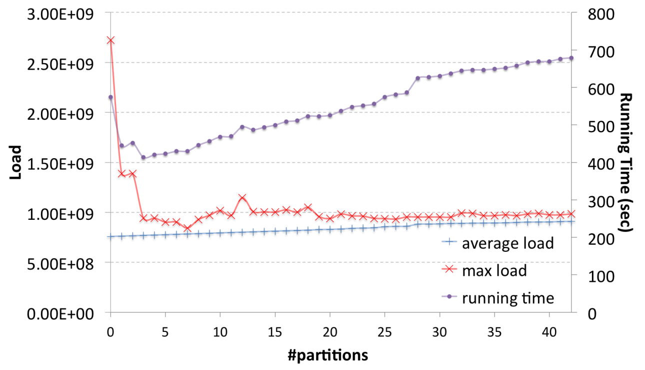

Figure 4: Max load, average load, and estimated running time for data set Zipf-5m-[1,0].

Algorithm analysis: In each iteration of the while-loop, Alg. 5 computes I_t and $\mathcal{R}$ in constant time. (I_t can be computed incrementally by simply adding $|S_a|\sigma_S$ or $|T_a|\sigma_T$, respectively.) For I_m and O_m, let $k = \sum_{h\in H} p_h q_h$ denote the number of (sub) groups. Using LLD load assignment, sorting of (sub) groups by induced load dominates, resulting in O($k \log k$) complexity. Load assignment using k' randomly selected hash functions incurs asymptotic cost O($k'k$). The heavy hitter sub-group partitioning is found by a version of SimpleGreedy (Alg. 2), which we refer to as **SimpleGreedy-R**. It calls GreedyPartition-R, instead of GreedyPartition. Hence complexity compared to the counterpart without $\mathcal{R}$ increases by a factor $k \log k$ for LLD and $k'k$ when using k' hash functions.

6 DISCUSSION

This section explores the impact of our results from the practitioner's point of view.

Submodularity for foreign-key joins. Load variance minimization is submodular for *all* foreign-key join problems (Theorem 4.6). In fact, for foreign-key joins, Alg. 1 behaves in a very intuitive way. W.l.o.g. let T be the relation where A is the key. Then the cost of incrementing any p_h is always 1, no matter which S_h is partitioned. Since benefit is equal to $\frac{(\gamma+\alpha)^2|S_h|^2}{p_h(p_h+1)} - \gamma^2$ (see Eq. 19 in Appendix A.2), the S_h with the greatest value of $\frac{|S_h|^2}{p_h(p_h+1)}$ is partitioned next. Since $\frac{|S_h|^2}{p_h(p_h+1)} \approx \frac{|S_h|^2}{p_h^2}$, this tends to be the heavy hitter whose sub-groups are currently the largest.

Submodularity in general. For other equi-join problems, submodularity depends on the ratio γ/α. Even if it does not hold, one can still use the greedy partitioning algorithm, we simply lose the near-optimality *guarantee*. Notice that even without submodularity, the algorithm may still be near-optimal in practice, but we cannot prove it any more. Fortunately, in our experience, submodularity tends to hold as the following discussion shows.

As shown in Section 4.2, for objective functions f_1 and f_2, and hence by construction $-\mathbb{V}[L]$, to be both monotone and submodular in all cases, it is *sufficient* (though not necessary) that $|S_h|/(p_h+1) \geq \sqrt{2}\gamma/\alpha$ and, for rectangular partitioning, also $|T_h|/(q_h+1) \geq \sqrt{2}\gamma/\alpha$. Recall that γ/α captures the ratio of per-input-tuple processing time to per-output-tuple processing time. This ratio does not necessarily have to be near 1. For example, in MapReduce and Spark, due to backup-copies in the distributed file system, reading might be significantly cheaper *per tuple* than writing. (The default is to make three copies of each file chunk in HDFS.) In that case, γ/α may be significantly smaller than 1. On the other hand, if join output is immediately consumed by an aggregate operator (instead of writing the full result out), then it might be significantly greater than 1.

To understand when $|S_h|/(p_h + 1) \geq \sqrt{2}\gamma/\alpha$ might not hold, recall that ratio $|S_h|/p_h$ denotes the number of input tuples from group S_h assigned to each partition (and analogously for $|T_h|/q_h$). We argue that a partitioning with fewer than 1000 input tuples per sub-group does not need to be considered in practice. Even on a low-end laptop, the straightforward nested-loop implementation—load from disk 1000 tuples from S_h and 1000 tuples from T_h, store them in an array and then loop through all pairs—completed in less than a second for realistic tuple sizes, e.g., from the TPC benchmarks. When distributing pieces of work equivalent to 1 second of processing time, uneven load distribution is of no concern in a distributed system where even starting up a job takes minutes. Hence a more fine-grained partitioning can only achieve insignificant load *balance* improvement.

This implies submodularity for a wide range of γ/α ratios. For $|S_h|/p_h \geq 1000$, and using that $p_h \geq 1$ implies $p_h + 1 \leq 2p_h$, we obtain $|S_h|/(p_h+1) \geq 500$. Hence for $|S_h|/(p_h+1) \geq \sqrt{2}\gamma/\alpha$ to hold, it is sufficient that $\gamma/\alpha \leq 353$. In all our experiments (Section 7), which include the most aggressive output-aggregation queries like SELECT COUNT(*) FROM S, T WHERE S.A=T.A, and very large input tuples with 1657 attributes, the ratio γ/α was always well below 350. In summary, we generally expect both monotonicity and submodularity to hold for all sub-group partitionings that *will be considered during the optimization process in practice.*

Random load assignment. Consider a user with skewed data who would like to use a distributed relational DBMS, Hive [35] or Spark SQL. These systems all offer implementations of the classic parallel hash join. Instead of forcing users to change the built-in hash join implementations to make them skew-resistant, our techniques can be applied to *pre-process* the input so that even the built-in default hash join will not suffer from skew-related processing delays. More precisely, after our technique identifies the near-optimal partitioning, a simple one-pass parallel scan can add sub-group keys to the join-attribute value of tuples in partitioned groups. This algorithm is equivalent to the Map program in Alg. 3. Then the classic hash join will work with the new keys, distributing the *sub*-groups over the workers.

Deterministic load assignment. In our experience, LLD tends to find better load assignments than random (hashing). Since the default parallel join implementations we are aware of use hashing and not LLD, the user would have to provide his/her own implementation. If s/he does, then, as discussed in Section 5, we can still use the greedy partitioning that was developed for random load assignment. It loses the near-optimality guarantees, hence turns into "yet another heuristic". Interestingly, as our experiments show, this heuristic beats the state of the art across different join types, clusters, and diverse data sets. We intend to investigate this in depth in future work. Note also that *no existing algorithm provides practically relevant near-optimality guarantees* for running time. In particular, even the asymptotically near-optimal BKS algorithm (see Section 7) falls short in several ways. (1) It only considers asymptotic cost, hence might be off by a large constant factor compared to optimal. (2) It does not support triangular partitioning, thus introducing a much higher data duplication cost for natural self-joins. And (3), it only considers input-induced load.

7 EXPERIMENTAL EVALUATION

The previous sections proved analytically that we can efficiently find near-optimal combinations of load expectation and variance. Hence the main goal of the experiments is to show that optimizing for this tradeoff results in competitive running times for the join partitioning found. In all cases, computation time of Alg. 5 for finding the sub-group partitioning was negligible (less than 1 sec on a single machine) compared to running time of the join itself. Each experiment was repeated multiple times; since running times varied by less than 10%, we omit error bars and only report averages.

7.1 Basic Setup

Environments. We implemented all algorithms in Hadoop MapReduce and conducted experiments on two different systems. `Cluster14` is an in-house cluster consisting of eight machines (quad-core Xeon 2.4GHz processor, 8GB RAM, 500GB SATA disk, Linux) running Hadoop 1.2, connected by a Gigabit network switch. One machine is dedicated as the master, leaving 7 machines for a total of 14 workers (2 cores are used on each machine). `Emr50` consists of 51 virtual machines of type m1.medium (1 virtual CPU, 3.75GB RAM, 410GB disk, "moderate" network performance) on Amazon's Elastic MapReduce cloud. One machine is dedicated as the master, while the other 50 serve as workers. This cluster runs Hadoop 2.7.3 with the YARN scheduler in default configuration. Jobs on both clusters read and write to HDFS.

Queries and data are selected to cover a wide variety of degrees of skew, γ/α ratios (by having a join with and without aggregation and by varying the number of attributes in the input relations), and different bottlenecks (by varying the number of tuples and tuple size, the size of the largest join group, and the number of small groups).

Queries. `JOIN` computes the full equi-join, emitting all result tuples. `JOIN-AGG` computes an equi-join whose results are aggregated on-the-fly as they are generated, resulting in significantly lower α. (We compute the sum over a non-join attribute.) Only a single output tuple is emitted for each join group. `SELF-JOIN` and `SELF-JOIN-AGG` are self-join versions of `JOIN` and `JOIN-AGG`, respectively.

Data. We show representative results on a variety of synthetic and real data, summarized in Table 2.

`Zipf`-n-z denotes a pair of synthetic data sets with Zipf-distributed join attribute, with skew parameter z. If the two inputs have different z, we include both, e.g., Zipf-5m-[1,0] indicates that one data set has $z = 1$, the other $z = 0$. (As usual, $z = 0$ results in uniform distribution; values between 0.25 and 1.0 represent degrees of skew often observed in practice.) Each data set contains n tuples with join attribute values between 1 and 20. By default, join attribute distribution is correlated in the sense that the most frequent value in one input is also the most frequent in the other. In

Data sets	|I| ($\times 10^9$)	|O| ($\times 10^9$)	**Number of Columns**
Zipf-100k-1.0-1g*	1.00	0.01	2
Zipf-500k-1.0-1g*	1.00	0.1	2
Zipf-90k-1.0-1g	1.00	1	2
Zipf-200k-1.0-50m	0.05	5	2
Zipf-285k-1.0-10m	0.01	10	2
Zipf-100k-1.0	0.0002	1.23	2
Zipf-500k-0.25	0.001	13.11	2
Zipf-500k-0.5	0.001	15.59	2
Zipf-500k-1.0	0.001	30	2
Zipf-5m-0	0.01	1250	200
Zipf-5m-[0.5,0]	0.01	1250	200
Zipf-5m-[1,0]	0.01	1250	200
Zipf-5m-0.5	0.01	1559	200
Zipf-5m-1.0	0.01	3083	200
ebird-basic	0.0038	600	953
ebird-all	0.0038	600	1657
cloud-200k	0.0004	6.52	4
cloud-5m	0.01	4077	100
cloud-10m	0.02	16353	100
tpc-h-cust-nation	0.2	0.2	8

Table 2: Data set properties

order to create various input-output size ratios, we selectively apply two modifications: (1) Add to Zipf-n-z many small groups with join attribute values randomly sampled from 100 to 2,000,000, which significantly increases the input size while virtually not affecting output. These data sets are named Zipf-n-z-s, where s denotes the total number of tuples in each input. (2) Remove the correlation between the Zipf distributions of the two inputs, i.e., the frequent join attribute values in one input may be frequent or infrequent in the other. These data set pairs have a "*" added to their name.

`cloud-n` denotes a pair of real data sets containing n tuples randomly sampled from a set of cloud reports [14]. They are joined on latitude, which was quantized into 10 equi-width bins to model a climate-zone based correlation analysis.

`ebird-all` is another real data set containing 1.89 million bird sightings, each with 1657 attributes describing properties of observation event, climate, landscape, etc [26]. `ebird-basic` is the same set, but with only the 953 most important columns. For both eBird data sets, we compute the self-join on three Boolean attributes, capturing presence (yes or no) of the top-3 most frequently reported bird species in North America. This was motivated by correlation studies exploring habitat properties based on species appearance patterns.

`tpc-h-cust-nation` is from TPC-H [36] and we run foreign-key join between tables CUSTOMER and NATION on "NATIONKEY". This join shows up in Q7, Q8 and Q10. NATION has only 25 tuples as there are 25 distinct NATIONKEY values according to the TPC-H spec. To add realistic data skew, we make the foreign key CUSTOMER.NATIONKEY follow a Zipfian distribution with $z = 1.0$.

Algorithms. We compare end-to-end running time against state-of-the-art competitors.

`ExpVar`: our proposed SimpleGreedy algorithm (Alg. 2). It calls Alg. 5, using LLD assignment and considering all groups as heavy-hitter candidates.

`NoPar`: the standard hash-based join algorithm that does not split groups into sub-groups.

`PaBr`: the "partition and broadcast" algorithm. For each heavy hitter $h \in H$, the larger of S_h and T_h is partitioned into w equal chunks (one assigned to each worker), while the smaller is broadcast to all workers. In previous work, heavy hitters are often defined as those groups that have greater than average load. We ran three different versions of the algorithm, each for a different definition of load: considering only input, only output, or the sum of input and output. The reported results are for the best of the three, which in most cases were the latter two.

`PaBr+`: our improved version of `PaBr` that uses our running-time model $\mathcal{R}$ (Section 5.2) to determine the best heavy-hitter threshold. The algorithm sorts all groups by load, then tries each possible threshold—from the load of the biggest group to that of the smallest—and returns the partitioning with the shortest estimated running time.

`BKS`: the improved hypercube-based algorithm that was shown to be asymptotically near-optimal for binary equi-joins [2]. For any heavy hitter $h \in H$, it assigns $w_h = \lceil w \cdot \frac{|S_h|}{\sum_{i \in H} |S_i|} \rceil + \lceil w \cdot \frac{|T_h|}{\sum_{i \in H} |T_i|} \rceil + \lceil w \cdot \frac{|S_h||T_h|}{\sum_{i \in H} |S_i||T_i|} \rceil$ workers to process join group h. Given w_h, a rectangular partitioning of w_h sub-groups is created. Heavy hitters are defined as those groups whose load is above average.

`BKS+`: our improved version of `BKS` that, like `PaBr+`, uses our cost model to determine the best heavy hitter threshold.

`ExpVarTri`: our proposed algorithm, a version of SimpleGreedy calling Alg. 5, using the triangular partitioning for self-joins.

`CIK`: a specialized sub-group partitioning algorithm for self-joins, adapted from the state-of-the-art "Dis-Dedup" algorithm [6]. The number of workers assigned to each heavy-hitter group is proportional to its load. Details are described in Appendix B.1. The only difference is that for $i \neq j$ and $s_i, s_j \in S_h$, Dis-Dedup only outputs either (s_i, s_j) or (s_j, s_i), but not both. (Duplication checking is symmetric, i.e., only one of the two is needed.) For a self-join, both pairs have to be emitted.

`CIK+`: our improved version of `CIK` that, like `PaBr+`, uses our cost model to determine the best heavy hitter threshold.

Note that different algorithms may find the same sub-group partitioning, which results in identical running time numbers in the tables. We always use the best load assignment for a given set of (sub) groups, even if the originally proposed version of the algorithm used a worse one. This way all reported differences are due to the quality of the sub-group partitioning found. The only exception is the `NoPar` algorithm that does not create sub-groups and by definition uses a simple hash-based load assignment.

Similarly, we use the same running-time cost model $\mathcal{R}$ for any cost estimation done by any of the algorithms. The running-time models for a cluster are trained using a total of only 50 training points. They are selected such that a space of n^3 possible (I_t, I_m, O_m) combinations contains only $O((\log_{10} n)^3)$ training points—to ensure a reasonable offline training investment even if input and output of different joins processed on a given cluster differ by several orders of magnitude.

Heavy hitter statistics. We made count statistics for all groups, not just the heavy hitters, available to all algorithms. This was done

Data sets	ExpVar	NoPar	PaBr	PaBr+	BKS	BKS+
Zipf-200k-1.0-50m	755	5005	1137	769	1181	913
Zipf-285k-1.0-10m	1352	7965	2160	1324	1507	1448
Zipf-100k-1.0	208	1270	299	201	240	217
Zipf-500k-0.25	1765	4599	1937	1756	3199	1808
Zipf-500k-0.5	2125	7676	3663	2048	3668	2535
Zipf-500k-1.0	4033	27455	5389	4114	5338	4310
Zipf-100k-1.0-1g*	1570	1837	1764	1660	1789	1789
Zipf-500k-1.0-1g*	1661	1961	1968	1728	1817	1817
Zipf-90k-1.0-1g	1789	3198	2059	1900	1873	1869
cloud-200k	1144	4448	1612	1204	1594	1215
tpc-h-cust-nation	433	1294	448	448	597	467

Table 3: Running times (sec) for JOIN on Cluster14

Data sets	ExpVarTri	CIK	CIK+
sj-zipf-200k-1.0-50m	843	1434	1395
sj-zipf-500k-0.25	1923	2827	2827
sj-cloud-200k	1220	2263	2263

Table 4: Running times (sec) for SELF-JOIN on Cluster14

for several reasons. (1) Even if those statistics are not available or are incomplete, it is cheap to collect them compared to the cost of a join. (In our experiments, it took less than 1/10 of the time of the fastest join implementation; for JOIN much less.) (2) There is no universally accepted definition of a heavy hitter. ExpVar *does not need a pre-defined heavy-hitter threshold*. Its variance/duplication-ratio based optimization automatically determines which groups to partition—generally the largest ones first. On the other hand, previous work requires a given threshold. Proposed thresholds turned out to be suboptimal in many cases, therefore we included the corresponding "+" versions of the algorithms. They use the same cost model as our approach in order to make a better selection automatically.

7.2 Results for JOIN

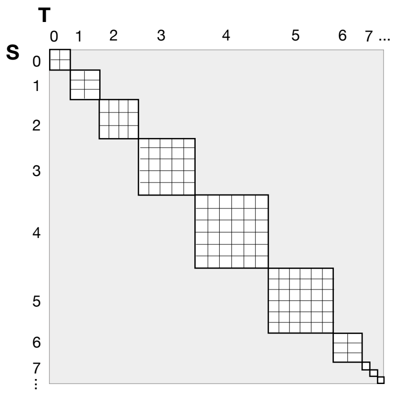

Figure 5: ExpVar partitioning for JOIN on cloud-200k

Table 3 shows the running times for computing JOIN on Cluster14. ExpVar leads to the lowest running-time on all data sets, no matter if join input is larger or smaller than output. Its partitioning, shown for an example in Figure 5, automatically identifies

Data sets	ExpVar	NoPar	PaBr	PaBr+	BKS	BKS+
cloud-5m	314	1291	473	446	329	325
ebird-all	1367	3000	3222	3000	1927	1871
ebird-basic	598	1051	1842	1051	939	907
Zipf-5m-0.5	353	591	693	498	494	454
Zipf-5m-1.0	562	1502	989	924	692	643
Zipf-5m-0	333	378	378	325	378	365
Zipf-5m-[0.5,0]	321	418	474	304	391	379
Zipf-5m-[1,0]	351	495	894	474	461	359

(a) Running times (sec) on Cluster14.

Data sets	ExpVar	NoPar	PaBr	PaBr+	BKS	BKS+
cloud-5m	440	3652	1088	1088	546	546
ebird-basic	380	3410	1022	684	464	425
BKS-bad-case	2072	3212	5419	2072	3574	2072

(b) Running times (sec) on Emr50.

Table 5: Performance for JOIN-AGG.

which groups to partition and how much. NoPar generally performs the worst due to poor load balancing caused by under-partitioning. ExpVar beats the mainstream algorithm PaBr by a large margin in most cases. PaBr, and in some cases also BKS, under-partition "medium" hitters whose size is just below the heavy-hitter threshold, resulting in poor load balance. PaBr+ and BKS+ perform better, because they select a lower threshold based on the running-time model.

The last line in Table 3 shows the case of a foreign-key join between CUSTOMER and NATION. Since the join is on NATION's primary key, only CUSTOMER gets partitioned. Hence any algorithm that properly partitions and distributes CUSTOMER in a balanced way should achieve a competitive running time. Only NoPar and BKS suffer from poor load balance due to under-partitioning.

Overall, all algorithms that use a cost model to guide the partitioning process (ExpVar, PaBr+, BKS+) perform similarly well, because they avoid under-partitioning of large join groups. It turns out that moderate over-partitioning (and hence "unnecessary" input duplicates) *for large join groups* has negligible effect on running time due to the very high output-to-input size ratio. This will be different for JOIN-AGG.

The running times for computing SELF-JOIN are reported in Table 4. ExpVarTri beats the other two algorithms by a large margin of up to 80%. CIK suffers from under-partitioning on large groups compared to our ExpVarTri algorithm. CIK+ sometimes improves on CIK slightly, but still suffers from the same problem.

7.3 Results for JOIN-AGG

This set of experiments explores running time for the join that immediately aggregates its output, i.e., emits only a small aggregate result of a few dozen result tuples—a single tuple per join group. Note that the larger cluster takes longer on the same problem instances, because of the slower network and slower individual workers. The slower individual machines affected local running time of the most loaded worker for NoPar, while the slower network delayed shuffling for the large inputs for the other algorithms.

Tables 5a and 5b show representative results on the two clusters. Our proposed ExpVar technique wins in all cases, sometimes by a wide margin, no matter how skewed the input. NoPar suffers from the high load induced by the largest input group on the worker assigned to it. PaBr and PaBr+ tend to over-partition large groups, but under-partition slightly smaller ones, because of their simplistic binary approach of either partitioning into w sub-groups (for heavy hitter groups) or not partitioning at all (all other groups).

BKS, which also relies on a heavy-hitter threshold, performs well. This is caused by a more intelligent partitioning of heavy hitters, nearly eliminating the effect of over-partitioning. However, it still suffers from under-partitioning when the number of groups is small and those "medium" hitters are not split. To further explore this aspect, we created BKS-bad-case as follows. The input consists of 49 groups of size $1.2 \cdot 10^6$ by $2 \cdot 10^5$ each, and a single group of size $1.19 \cdot 10^6$ by $1.9 \cdot 10^5$. The former are above the average, the latter is below, and hence will not be split by BKS. Even though it finds a near-optimal partitioning for the 49 bigger groups (each is partitioned into 6 sub-groups of size 200,000 by 200,000), the small group is not partitioned at all. The worker receiving this group ends up delaying job completion, even though all (sub) groups are optimally distributed over the 50 workers. BKS+ finds the optimal solution thanks to our proposed extension of leveraging a cost model for identifying the heavy hitter threshold.

ExpVar and PaBr+ automatically find the optimal solution, i.e., do not split any group, assigning one per worker. NoPar also does not split groups, but since it uses a hash function to assign groups to workers (instead of LLD, which is a better choice here), a worker might receive more than one, while another receives none.

The SELF-JOIN-AGG results are listed in Table 6. For all data sets, ExpVarTri wins over the competitors. Similar to the SELF-JOIN case, CIK+ in some cases improves the running time over CIK, but not by a large margin.

7.4 A Look Under the Hood

Tables 7 and 8 show detailed statistics for some of the data sets. In Table 7, the first data set has larger input (2 billion tuples) than output, and very few large groups. The largest of them has only about 25K tuples in each S and T. Despite being so small, that group produces about 6.25E+8 output tuples, almost ten times the average per-worker output of $10^9/14$ = 7.1E+7. Hence the right strategy for this data set is to carefully partition the few largest groups. All but NoPar and PaBr do this to some degree, but ExpVar and PaBr+ stand out by coming close to the ideal output per worker of 7.1E+7, while generating negligible input duplication.

For the second data set, output is four orders of magnitude greater than input. Hence the right approach for running-time minimization is to aggressively balance output load, even at the cost of high input duplication. ExpVar automatically recognizes this, going further than the competitors in partitioning medium-sized groups and hence winning on running time (Table 3).

For JOIN-AGG, output size never exceeds a few dozen tuples (one aggregate per join group), hence cost per input tuple is significantly higher than per output tuple. Compared to JOIN, the winning partitioning strategy therefore should reduce input duplication, while paying less attention to load balancing. However, there are some large groups that produce very high *computation cost* as indicated by the large number of joined pairs in the last column of Table 8. On cloud-5m, ExpVar strikes the best balance by producing significantly lower maximal input per worker than all competitors. Only NoPar produces fewer input duplicates in *total*, but it suffers from the high computation cost for the biggest join group. On ebird-all, interestingly the BKS algorithms beat ExpVar on *both* max input and output *per worker*, i.e., do better in terms of minimizing max worker load. However, our approach recognizes correctly that here the total input duplication matters more due to high communication cost relative to computation cost. By generating less than half the total amount of input, ExpVar significantly beats the BKS approaches.

8 RELATED WORK

Hash partitioning has been used for equi-joins since the dawn of parallel databases [9, 20]. In Hive, the optimizer may select a join implementation where one input is partitioned into chunks arbitrarily, while the other is copied to every worker [35].

The limiting effect of join attribute skew on speedup is well known and was discussed by Walton et al. [39]. Some earlier works [17, 19] propose methods to distribute groups more evenly. While some techniques assign load statically, e.g., by using bin packing or scheduling heuristics such as first-fit decreasing [17], others attempt to dynamically remedy load imbalance at runtime [8, 12, 15, 30]. Our work is orthogonal to the choice of static or dynamic load balancing, i.e., the proposed partitioning could be applied in either case as soon as (approximate) count statistics per input are known.

In most of those earlier works, splitting of groups was not considered. Later techniques rely on a threshold to identify "heavy hitter" groups and typically use partition-broadcast for them: the larger input is partitioned over all workers, while the smaller is broadcast [28, 41, 42]. As our experiments show, the resulting over-partitioning of heavy hitters and under-partitioning of the other groups can lead to poor performance. Granularity of group partitioning was left as a user-defined parameter or driven by factors such as number of workers or memory size [10, 23, 29]. Bruno et al. [4] consider assigning a subset of workers to each heavy hitter, but partition granularity is determined by a simplistic per-group cost analysis that does not take other groups into account. The state of the art algorithm determining how many workers to assign to each heavy hitter group is the asymptotically near-optimal algorithm by Beame et al. [2], which is included in our experimental comparison (BKS).

Input partitioning was also considered for the more general problem of distributed theta-join computation. Vitorovic et al. [38] propose a new tiling algorithm to partition the join matrix in a balanced way, improving over earlier work by Okcan and Riedewald [27]. However, the authors themselves point out that for equi-joins one should instead rely on a specialized solution such as [2], because general theta-join approaches do not exploit the strong structural properties of key-equality based matching in equi-joins.

For kNN and similarity joins in distributed systems, various partitioning-based approaches have been proposed. Zhang et al. [43] employ z-value based sorting of data and range-partition accordingly, which allows the search of (approximately) top-k

Data sets	ExpVarTri	CIK	CIK+
cloud-5m	306	541	541
ebird-all	607	696	607

(a) Running times (sec) on `Cluster14`

Data sets	ExpVarTri	CIK	CIK+
cloud-10m	808	1148	1148

(b) Running times (sec) on `Emr50`

Table 6: Performance for `SELF-JOIN-AGG`

Data sets	Alg.	#input tuples	#shuffled tuples	#input tuples on most loaded worker	#result tuples on most loaded worker	#result tuples
Zipf-90k-1.0-1g	ExpVar	2.00E+09	2000896412	71494429	72541536	9.99E+08
Zipf-90k-1.0-1g	NoPar	2.00E+09	2000000000	71428572	625750225	9.99E+08
Zipf-90k-1.0-1g	PaBr	2.00E+09	2000179976	71478602	625750225	9.99E+08
Zipf-90k-1.0-1g	PaBr+	2.00E+09	2001023078	71508779	73098595	9.99E+08
Zipf-90k-1.0-1g	BKS	2.00E+09	2000428372	71461349	89873111	9.99E+08
Zipf-90k-1.0-1g	BKS+	2.00E+09	2000546846	71479838	87776981	9.99E+08
Zipf-500k-1.0	ExpVar	1.00E+06	8.43E+06	6.11E+05	2.21E+09	3.08E+10
Zipf-500k-1.0	NoPar	1.00E+06	1.00E+06	2.78E+05	1.93E+10	3.08E+10
Zipf-500k-1.0	PaBr	1.00E+06	4.31E+06	3.92E+05	3.24E+09	3.08E+10
Zipf-500k-1.0	PaBr+	1.00E+06	5.68E+06	4.46E+05	2.26E+09	3.08E+10
Zipf-500k-1.0	BKS	1.00E+06	2.41E+06	2.25E+05	3.30E+09	3.08E+10
Zipf-500k-1.0	BKS+	1.00E+06	2.51E+06	1.90E+05	2.39E+09	3.08E+10

Table 7: `JOIN` on `Cluster14`: detailed statistics

Data sets	Alg.	#input tuples	#shuffled tuples	#input tuples on most loaded worker	#joined pairs on most loaded worker
cloud-5m	ExpVar	1E+07	1.53E+07	1.31E+06	3.81E+11
cloud-5m	NoPar	1E+07	1.00E+07	2.28E+06	1.30E+12
cloud-5m	PaBr	1E+07	6.97E+07	5.57E+06	3.87E+11
cloud-5m	PaBr+	1E+07	4.99E+07	4.42E+06	4.67E+11
cloud-5m	BKS	1E+07	2.73E+07	2.40E+06	3.32E+11
cloud-5m	BKS+	1E+07	2.57E+07	2.24E+06	3.05E+11
ebird-all	ExpVar	1.89E+06	4.44E+06	9.85E+05	2.16E+11
ebird-all	NoPar	1.89E+06	1.89E+06	1.32E+06	4.34E+11
ebird-all	PaBr	1.89E+06	2.68E+07	2.14E+06	6.09E+10
ebird-all	PaBr+	1.89E+06	6.64E+06	3.94E+06	4.31E+11
ebird-all	BKS	1.89E+06	1.06E+07	8.57E+05	5.64E+10
ebird-all	BKS+	1.89E+06	1.06E+07	8.57E+05	5.64E+10

Table 8: `JOIN-AGG` on `Cluster14`: detailed statistics

neighbors for each record within a small range. Lu et al. [24] exploit Voronoi diagram-based partitioning so that kNN join can be answered by checking data points within each partition. For set-similarity joins, Vernica et a. [37] use prefix filters and hash-partition input data onto different workers. For similarity joins with edit distances, Jiang et al. [18] design parallel algorithms using multi-core processors. For similarity joins on general metric distance, [33] and [40] propose algorithms that partition input data sets into sufficiently small subsets in an ad-hoc manner. Sarma et al. [7] design a dynamic partitioning scheme that can balance load distribution. Tang et al. [34] focus on similarity joins of tree-structured objects, and propose a novel partitioning approach that can decompose tree objects into balanced subgraphs. For all these approaches for kNN and similarity joins, the main contributions are centered around pruning of non-joinable pairs, ways to map kNN/similarity joins to equi-join like hash partitioning, and partitioning methods for processing such joins in parallel. These works are orthogonal to ours.

Several previous publications experimentally compare distributed join performance [3, 5, 9, 31, 32]. Afrati and Ullman propose a cost analysis to minimize communication cost, assuming even load distribution for running time estimation [1]. Kwon et al. [22] uses dynamic scheduling to mitigate the impact of skewness in MapReduce programs, but cannot automatically split groups as it treats the map and reduce functions as black boxes. Recent work on storage layout like AdaptDB [25] focuses on optimizing communication cost for workloads with various join queries on different attributes. Duggan et al. [11] propose a skew-aware join optimization framework for array databases.

9 CONCLUSIONS

This work presents an interesting and novel way to approach distributed join computation. Previous work either falls into the category of heuristics without any optimality guarantees or optimality guarantees for *asymptotic* cost. By quantifying the tradeoff between load expectation and variance, we were able to design novel algorithms that provide constant-factor approximation compared to an optimal solution. More precisely, we identified algorithmically verifiable conditions that are sufficient for ensuring that minimizing load variance subject to a limit on load expectation is a monotone submodular maximization problem with Knapsack constraints. Our results apply to all equi-join problems, including self-joins and the most general load definition as the weighted sum of input and output. Monotonicity and submodularity hold for a wide variety of practically relevant ratios of per-input-tuple to per-output-tuple processing time.

For cases where submodularity does not hold or when using deterministic load assignment, optimality cannot be proven any more. Nevertheless, extensive experiments showed that sub-group partitioning driven by the expectation-variance tradeoff for random load assignment works very well for deterministic load assignment in practice, beating the state of the art across a wide variety of problems. In future work we will explore how to generalize the approach to multi-way joins and how to include deterministic load assignment in the analytical results.

10 ACKNOWLEDGMENTS

This work is supported by the National Institute of Health (NIH) under award number R01 NS091421 and by a Northeastern University (NU) Tier 1 award. The content is solely the responsibility of the authors and does not necessarily represent the official views of NU or NIH. The authors also would like to thank the reviewers for their constructive feedback.

REFERENCES

[1] Foto N Afrati and Jeffrey D Ullman. 2010. Optimizing joins in a map-reduce environment. In *the 13th International Conference on Extending Database Technology*. 99–110.

[2] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in parallel query processing. In *PODS*. 212–223.

[3] Spyros Blanas, Jignesh M Patel, Vuk Ercegovac, Jun Rao, Eugene J Shekita, and Yuanyuan Tian. 2010. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*. 975–986.

[4] Nicolas Bruno, YongChul Kwon, and Ming-Chuan Wu. 2014. Advanced join strategies for large-scale distributed computation. *VLDB* (2014), 1484–1495.

[5] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*. 63–78.

[6] Xu Chu, Ihab F Ilyas, and Paraschos Koutris. 2016. Distributed data deduplication. *VLDB* 9, 11 (2016), 864–875.

[7] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. 2014. Clusterjoin: a similarity joins framework using map-reduce. In *VLDB*. 1059–1070.

[8] Hasanat M Dewan, Kui W Mok, Mauricio Hernández, and Salvatore J Stolfo. 1994. Predictive dynamic load balancing of parallel hash-joins over heterogeneous processors in the presence of data skew. In *PDIS*. 40–49.

[9] David J DeWitt and Robert Gerber. 1985. Multiprocessor hash-based join algorithms. In *VLDB*. 151–164.

[10] David J DeWitt, Jeffrey F Naughton, Donovan A Schneider, and Srinivasan Seshadri. 1992. Practical skew handling in parallel joins. In *VLDB*. 27–40.

[11] Jennie Duggan, Olga Papaemmanouil, Leilani Battle, and Michael Stonebraker. 2015. Skew-aware join optimization for array databases. In *SIGMOD*. 123–135.

[12] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and adaptive online joins. *VLDB* (2014), 441–452.

[13] Ronald L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* 17, 2 (1969), 416–429.

[14] CJ Hahn, SG Warren, and R Eastman. 2012. Extended Edited Synoptic Cloud Reports from Ships and Land Stations Over the Globe, 1952-2009 (NDP-026C). (2012).

[15] Lilian Harada and Masaru Kitsuregawa. 1995. Dynamic Join Product Skew Handling for Hash-Joins in Shared-Nothing Database Systems.. In *DASFAA*. 246–255.

[16] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *VLDB* 4, 11 (2011), 1111–1122.

[17] Kien A Hua and Chiang Lee. 1991. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning.. In *VLDB*. 525–535.

[18] Yu Jiang, Dong Deng, Jiannan Wang, Guoliang Li, and Jianhua Feng. 2013. Efficient Parallel Partition-based Algorithms for Similarity Search and Join with Edit Distance Constraints. In *EDBT/ICDT Workshops*. 341–348.

[19] Masaru Kitsuregawa and Yasushi Ogawa. 1990. Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC).. In *VLDB*. 210–221.

[20] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. 1983. Application of hash to data base machine and its architecture. *New Generation Computing* 1, 1 (1983), 63–74.

[21] A. Krause and D. Golovin. 2013. *Tractability: Practical Approaches to Hard Problems.* Chapter Submodular Function Maximization.

[22] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD*. 25–36.

[23] HJ Lu and Kian-Lee Tan. 1994. Load-balanced join processing in shared-nothing systems. *J. Parallel and Distrib. Comput.* 23, 3 (1994), 382–398.

[24] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. 2012. Efficient processing of k nearest neighbor joins using mapreduce. *VLDB* 5, 10 (2012), 1016–1027.

[25] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. 2017. AdaptDB: Adaptive Partitioning for Distributed Joins. *VLDB* (2017), 589–600.

[26] Arthur M Munson, Kevin Webb, Daniel Sheldon, Daniel Fink, Wesley M Hochachka, Marshall Iliff, Mirek Riedewald, Daria Sorokina, Brian Sullivan, Christopher Wood, and Steve Kelling. 2014. The ebird reference dataset, version 2014. *Cornell Lab of Ornithology and National Audubon Society, Ithaca, NY* (2014).

[27] Alper Okcan and Mirek Riedewald. 2011. Processing Theta-joins Using MapReduce. In *SIGMOD*. 949–960.

[28] Orestis Polychroniou, Rajkumar Sen, and Kenneth A Ross. 2014. Track join: distributed joins with minimal network traffic. In *SIGMOD*. 1483–1494.

[29] Viswanath Poosala and Yannis E Ioannidis. 1996. Estimation of query-result distribution and its application in parallel-join load balancing. In *VLDB*. 448–459.

[30] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. 2016. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *ICDE*. 1194–1205.

[31] Donovan A Schneider and David J DeWitt. 1989. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *SIGMOD*. 110–121.

[32] Donovan A Schneider and David J DeWitt. 1990. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *VLDB*. 469–480.

[33] Yasin N. Silva and Jason M. Reed. 2012. Exploiting MapReduce-based Similarity Joins. In *SIGMOD*. 693–696.

[34] Yu Tang, Yilun Cai, and Nikos Mamoulis. 2015. Scaling Similarity Joins over Tree-structured Data. *VLDB* 8, 11 (2015), 1130–1141.

[35] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive-a petabyte scale data warehouse using hadoop. In *ICDE*. 996–1005.

[36] Transaction Processing Performance Council (TPC). 2017. TPC Benchmark H (Decision Support) Standard Specification. http://www.tpc.org. (2017).

[37] Rares Vernica, Michael J Carey, and Chen Li. 2010. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD*. 495–506.

[38] Aleksandar Vitorovic, Mohammed Elseidy, and Christoph Koch. 2016. Load balancing and skew resilience for parallel joins. In *ICDE*. 313–324.

[39] Christopher B Walton, Alfred G Dale, and Roy M Jenevein. 1991. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins.. In *VLDB*. 537–548.

[40] Ye Wang, Ahmed Metwally, and Srinivasan Parthasarathy. 2013. Scalable all-pairs similarity search in metric spaces. In *KDD*. 829–837.

[41] Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John Turek. 1994. New algorithms for parallelizing relational database joins in the presence of data skew. *TKDE* (1994), 990–997.

[42] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. 2008. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD*. 1043–1052.

[43] Chi Zhang, Feifei Li, and Jeffrey Jestes. 2012. Efficient parallel kNN joins for large data in MapReduce. In *EDBT*. ACM, 38–49.

A RECTANGULAR PARTITIONING: MONOTONICITY AND SUBMODULARITY

This section contains detailed proofs for the monotonicity and submodularity of $-\mathbb{V}[L]$ when rectangular partitioning is used, i.e., the corresponding theorems mentioned in Sections 4.1 and 4.2.

A.1 General Case

A.1.1 Monotonicity (Theorem 4.1). To prove Theorem 4.1, we first prove a stronger result as follows.

Lemma A.1. *Worker load variance $-\mathbb{V}[L]$ is monotonically increasing in the number of partitions p_h of S_h and q_h of T_h, for each join attribute value $h \in H$, if $\alpha \geq \left(\frac{\sqrt{p_h(p_h+1)}}{|S_h|} - \frac{q_h}{|T_h|}\right)\gamma$ and $\alpha \geq \left(\frac{\sqrt{q_h(q_h+1)}}{|T_h|} - \frac{p_h}{|S_h|}\right)\gamma$.*

Proof. Define

$$g_1(p_h, q_h) = -\frac{w^2\mathbb{V}[L]}{w-1} = -\sum_{k \in H} p_k q_k (\gamma \frac{|S_k|}{p_k} + \gamma \frac{|T_k|}{q_k} + \alpha \frac{|S_k||T_k|}{p_k q_k})^2.$$

Then the discrete derivative on p_h is

$$\begin{aligned} &g_1(p_h + 1, q_h) - g_1(p_h, q_h) = \\ &-\left[(p_h+1)q_h\left(\gamma\frac{|S_h|}{p_h+1} + \gamma\frac{|T_h|}{q_h} + \frac{\alpha|S_h||T_h|}{(p_h+1)q_h}\right)^2\right. \\ &\qquad \left. + \sum_{\substack{k \in H \\ k \neq h}} p_k q_k \left(\gamma\frac{|S_k|}{p_k} + \gamma\frac{|T_k|}{q_k} + \frac{\alpha|S_k||T_k|}{p_k q_k}\right)^2\right] \\ &\qquad + \sum_{k \in H} p_k q_k (\gamma\frac{|S_k|}{p_k} + \gamma\frac{|T_k|}{q_k} + \alpha\frac{|S_k||T_k|}{p_k q_k})^2 \\ &= \frac{\gamma^2(q_h^2|S_h|^2 - p_h(p_h+1)|T_h|^2) + \alpha^2|S_h|^2|T_h|^2 + 2\alpha\gamma q_h|S_h|^2|T_h|}{p_h(p_h+1)q_h}. \end{aligned}$$

Since $p_h(p_h+1)q_h > 0$, g_1 is monotonically increasing in p_h when

$$\gamma^2(q_h^2|S_h|^2 - p_h(p_h+1)|T_h|^2) + \alpha^2|S_h|^2|T_h|^2 + 2\alpha\gamma q_h|S_h|^2|T_h| \geq 0$$
$$\Leftrightarrow \left(\alpha|S_h||T_h| + \gamma(q_h|S_h| + \sqrt{p_h(p_h+1)}|T_h|)\right)$$
$$\cdot \left(\alpha|S_h||T_h| + \gamma(q_h|S_h| - \sqrt{p_h(p_h+1)}|T_h|)\right) \geq 0$$
$$\Leftrightarrow \alpha|S_h||T_h| + \gamma(q_h|S_h| - \sqrt{p_h(p_h+1)}|T_h|) \geq 0$$
$$\Leftrightarrow \alpha \geq \frac{\sqrt{p_h(p_h+1)}|T_h| - q_h|S_h|}{|S_h||T_h|}\gamma = \left(\frac{\sqrt{p_h(p_h+1)}}{|S_h|} - \frac{q_h}{|T_h|}\right)\gamma \tag{12}$$

Similarly, the discrete derivative of g_1 on q_h is

$$g_1(p_h, q_h+1) - g_1(p_h, q_h)$$
$$= -\left[p_h(q_h+1)\left(\gamma\frac{|S_h|}{p_h} + \gamma\frac{|T_h|}{q_h+1} + \frac{\alpha|S_h||T_h|}{p_h(q_h+1)}\right)^2\right.$$
$$\left.+ \sum_{\substack{k\in H\\k\neq h}} p_k q_k\left(\gamma\frac{|S_k|}{p_k} + \gamma\frac{|T_k|}{q_k} + \frac{\alpha|S_k||T_k|}{p_k q_k)}\right)^2\right]$$
$$+ \sum_{k\in H} p_k q_k\left(\gamma\frac{|S_k|}{p_k} + \gamma\frac{|T_k|}{q_k} + \frac{\alpha|S_k||T_k|}{p_k q_k}\right)^2$$
$$= \frac{\gamma^2(p_h^2|T_h|^2 - q_h(q_h+1)|S_h|^2) + \alpha^2|S_h|^2|T_h|^2 + 2\gamma\alpha p_h|S_h||T_h|^2}{p_h q_h(q_h+1)},$$

so g_1 is monotonically increasing in q_h when

$$\gamma^2(p_h^2|T_h|^2 - q_h(q_h+1)|S_h|^2) + \alpha^2|S_h|^2|T_h|^2 + 2\gamma\alpha p_h|S_h||T_h|^2 \geq 0$$
$$\Leftrightarrow (\gamma(p_h|T_h| + \sqrt{q_h(q_h+1)}|S_h|) + \alpha|S_h||T_h|)$$
$$\cdot (\gamma(p_h|T_h| - \sqrt{q_h(q_h+1)}|S_h|) + \alpha|S_h||T_h|) \geq 0$$
$$\Leftrightarrow \gamma(p_h|T_h| - \sqrt{q_h(q_h+1)}|S_h|) + \alpha|S_h||T_h| \geq 0$$
$$\Leftrightarrow \alpha \geq \frac{\sqrt{q_h(q_h+1)}|S_h| - p_h|T_h|}{|S_h||T_h|}\gamma = \left(\frac{\sqrt{q_h(q_h+1)}}{|T_h|} - \frac{p_h}{|S_h|}\right)\gamma. \tag{13}$$

Hence, when α and γ satisfy inequalities 12 and 13, $-\mathbb{V}[L]$ is monotonically increasing. □

Since $\alpha \geq \frac{p_h+1}{|S_h|}\gamma$ implies Inequality 12, and $\alpha \geq \frac{q_h+1}{|T_h|}\gamma$ implies Inequality 13, Theorem 4.1 is proved.

A.1.2 Submodularity (Theorem 4.5). We consider a stronger result than Theorem 4.5, as follows.

Lemma A.2. *The set function f_1 is submodular iff for $\forall h \in H$, $\alpha \geq \sqrt{\frac{q_h(q_h+1)}{|T_h|^2} + \frac{p_h(p_h+1)}{|S_h|^2}} \cdot \gamma$.*

Proof. To prove submodularity of f_1, we need to compare the discrete derivatives of f_1 for sets X and $Y \supseteq X$, i.e., we want to show the following inequality.

$$(f_1(X\cup\{x\}) - f_1(X)) - (f_1(Y\cup\{x\}) - f_1(Y)) \geq 0 \tag{14}$$

Note that $x = (h, 0, i)$ or $x = (h, 1, j)$ for some $h \in H$ and $1 \leq i \leq |S_h|$ or $1 \leq j \leq |T_h|$. And $x \notin Y$, which means adding x to X increases the number of partitions p_h of S_h or q_h of T_h.

Considering $x = (h, 0, i)$ first, we need to show Inequality 14 for two cases: (1) $Y - X = \{(h, 0, i')\}, 1 \leq i' \leq |S_h|$; (2) $Y - X = \{(h, 1, j')\}, 1 \leq j' \leq |T_h|$. Then the desired result for any $Y \supseteq X$ follows from proof by induction.

For case (1), $f_1(X\cup\{x\}) = f_1(Y)$, so

$$(f_1(X\cup\{x\}) - f_1(X)) - (f_1(Y\cup\{x\}) - f_1(Y))$$
$$= p_h q_h(\frac{\gamma|S_h|}{p_h} + \frac{\gamma|T_h|}{q_h} + \frac{\alpha|S_h||T_h|}{p_h q_h})^2$$
$$- 2(p_h+1)q_h(\frac{\gamma|S_h|}{p_h+1} + \frac{\gamma|T_h|}{q_h} + \frac{\alpha|S_h||T_h|}{(p_h+1)q_h})^2$$
$$+ (p_h+2)q_h(\frac{\gamma|S_h|}{p_h+2} + \frac{\gamma|T_h|}{q_h} + \frac{\alpha|S_h||T_h|}{(p_h+2)q_h})^2$$
$$= \frac{2\gamma^2 q_h^2|S_h|^2 + 4\gamma\alpha q_h|S_h|^2|T_h| + 2\alpha^2|S_h|^2|T_h|^2}{p_h(p_h+1)(p_h+2)q_h} \geq 0.$$

For case (2),

$$(f_1(X\cup\{x\}) - f_1(X)) - (f_1(Y\cup\{x\}) - f_1(Y))$$
$$= -(p_h+1)q_h(\frac{\gamma|S_h|}{p_h+1} + \frac{\gamma|T_h|}{q_h} + \frac{\alpha|S_h||T_h|}{(p_h+1)q_h})^2$$
$$+ p_h q_h(\frac{\gamma|S_h|}{p_h} + \frac{\gamma|T_h|}{q_h} + \frac{\alpha|S_h||T_h|}{p_h q_h})^2$$
$$+ (p_h+1)(q_h+1)(\frac{\gamma|S_h|}{p_h+1} + \frac{\gamma|T_h|}{q_h+1} + \frac{\alpha|S_h||T_h|}{(p_h+1)(q_h+1)})^2$$
$$- p_h(q_h+1)(\frac{\gamma|S_h|}{p_h} + \frac{\gamma|T_h|}{q_h+1} + \frac{\alpha|S_h||T_h|}{p_h(q_h+1)})^2$$
$$= \frac{-\gamma^2|S_h|^2}{p_h(p_h+1)} + \frac{-\gamma^2|T_h|^2}{q_h(q_h+1)} + \alpha^2 \cdot \frac{|S_h|^2|T_h|^2}{p_h(p_h+1)q_h(q_h+1)}. \tag{15}$$

Eq. 15 being non-negative is equivalent to

$$\alpha \geq \sqrt{\frac{q_h(q_h+1)}{|T_h|^2} + \frac{p_h(p_h+1)}{|S_h|^2}} \cdot \gamma. \tag{16}$$

Symmetrically, for the case of $x = (h, 0, j)$, we also obtain the same condition as Inequality 16, completing the proof of Lemma A.2. □

For Inequality 16 to hold, it is sufficient that the following inequalities hold.

$$\alpha \geq \frac{\sqrt{2}(q_h+1)}{|T_h|}\gamma, \tag{17}$$

$$\alpha \geq \frac{\sqrt{2}(p_h+1)}{|S_h|}\gamma. \tag{18}$$

This completes the proof of Theorem 4.5.

A.2 Special Case: Foreign Key Joins

Now we take a look at a subclass of equi-join problems that frequently happen in practice. W.l.o.g., assume T is the relation where join attribute A is the key. Then $|T_h| = 1$ for all $h \in H$. Therefore we cannot partition on T_h any further, leaving $q_h = 1$. In this case, the monotonicity and submodularity of $\mathbb{V}[L]$ holds for any $\alpha \in [0, \infty)$ and $\gamma \in [0, \infty)$, as described in Theorems 4.2 and 4.6.

A.2.1 Monotonicity (Theorem 4.2).

Proof. Define

$$g_2(p_h, q_h) = -\frac{w^2}{w-1}\mathbb{V}[L] = -\sum_{i\in H} p_i\Big(\frac{\gamma|S_i|}{p_i} + \gamma + \frac{\alpha|S_i|}{p_i}\Big)^2$$
$$= -\sum_{i\in H}\left(\frac{(\gamma+\alpha)^2|S_i|^2}{p_i} + 2\gamma(\gamma+\alpha)|S_i| + p_i\gamma^2\right)$$

Then

$$\begin{aligned} & g_2(p_h+1, q_h) - g_2(p_h, q_h) \\ &= \left(\frac{(\gamma+\alpha)^2|S_h|^2}{p_h} + 2\gamma(\gamma+\alpha)|S_h| + p_h\gamma^2\right) \\ &\quad - \left(\frac{(\gamma+\alpha)^2|S_h|^2}{p_h+1} + 2\gamma(\gamma+\alpha)|S_h| + (p_h+1)\gamma^2\right) \\ &= \frac{(\gamma+\alpha)^2|S_h|^2}{p_h(p_h+1)} - \gamma^2 \end{aligned} \tag{19}$$

Since $p_h < p_h+1 \le |S_h|$, $\frac{(\gamma+\alpha)^2|S_i|^2}{p_i(p_i+1)} \ge (\gamma+\alpha)^2$. Hence, Eq. 19 is non-negative, which means that $-\mathbb{V}[L]$ is monotonically increasing. □

A.2.2 Submodularity (Theorem 4.6).

Proof. Since $q_h = 1$, we only need to prove that Inequality 14 holds when $Y - X = \{(h, 0, i)\}, 1 \le i \le |S_h|$. From the proof of Lemma A.2, and let $|T_h| = 1, q_h = 1$, we have

$$(f_1(X\cup\{x\}) - f_1(X)) - (f_1(Y\cup\{x\}) - f_1(Y))$$
$$= \frac{2(\gamma+\alpha)^2|S_h|^2}{p_h(p_h+1)(p_h+2)} \ge 0.$$

Therefore, f_1 is submodular in both cases, proving Theorem 4.6. □

B SELF-JOINS: TRIANGULAR PARTITIONING

B.1 Worker Assignment

For self-joins, the partitioning algorithm proceeds in two main steps: (1) determine how many workers to assign to each heavy hitter join group; (2) if a group takes more than one worker, partition it using *triangular* partitioning, as described in Section 2.

For Step (1), there are multiple assignment strategies. Chu et al [6] proposed assigning workers to each group proportional to its load. Specifically, groups are categorized into two types by a threshold W_{th}. Say Group 1 has load W_1, and if $W_1 \ge W_{\text{th}}$, then it is a "multi-worker group", otherwise it is a "single-worker group". For the former, $\lfloor w \cdot \frac{W_1}{W_{\text{total}}} \rfloor$ workers will be assigned to process Group 1, where W_{total} is the sum of the loads of all multi-worker groups. For the latter, Group 1 will not be partitioned, i.e., all tuples from this group will be sent to the same worker. The workers are assigned to single-worker groups in a round-robin fashion; and multiple single-worker groups can be processed on the same worker. In [6], the average load of all groups is used as the threshold, which corresponds to the CIK algorithm in Section 7. We also design an improved version which uses our cost model to determine the optimal threshold, as described in Section 7 (algorithm CIK+).

Another way of determining the number of workers assigned to each group is using our greedy algorithm (Section 7, algorithm ExpVarTri), because of the monotone submodularity of load variance. Corresponding proofs are in the following subsections.

B.2 Monotonicity (Theorem 4.3)

Proof. Recall that the monotonicity of $-\mathbb{V}[L]$ is equivalent to that of f_2 (Eq. 10). Let $x = (h, 0, j)$ for some $h \in H$ and $1 \le j \le |S_h|$, then the discrete derivative of f_2 is

$$\begin{aligned} & f_2(X\cup\{x\}) - f_2(X) \\ &= \left(\alpha^2|S_h|^4\frac{2p_h-1}{p_h^3} + \gamma\alpha|S_h|^3\frac{4p_h-2}{p_h^2} + \gamma^2|S_h|^2\frac{2p_h-1}{p_h}\right) \\ &\quad - \left(\alpha^2|S_h|^4\frac{2p_h+1}{(p_h+1)^3} + \gamma\alpha|S_h|^3\frac{4p_h+2}{(p_h+1)^2} + \gamma^2|S_h|^2\frac{2p_h+1}{p_h+1}\right) \\ &= \frac{4p_h^3+3p_h^2-p_h-1}{p_h^3(p_h+1)^3}\alpha^2|S_h|^4 + \frac{(4p_h^2-2)\gamma\alpha|S_h|^3}{p_h^2(p_h+1)^2} + \frac{-\gamma^2|S_h|^2}{p_h(p_h+1)}. \end{aligned} \tag{20}$$

When $\alpha \ge \frac{p_h}{|S_h|}\gamma$, from Eq. 20 follows

$$\begin{aligned} & f_2(X\cup\{x\}) - f_2(X) \\ &\ge \frac{4p_h^3+3p_h^2-p_h-1}{p_h^3(p_h+1)^3}\Big(\frac{p_h}{|S_h|}\Big)^2\gamma^2|S_h|^4 + \frac{4p_h^2-2}{p_h^2(p_h+1)^2}\frac{p_h}{|S_h|}\gamma^2|S_h|^3 \\ &\quad + \frac{-1}{p_h(p_h+1)}\gamma^2|S_h|^2 \\ &= \left(\frac{4p_h^3+3p_h^2-p_h-1}{p_h(p_h+1)^3} + \frac{4p_h^2-2}{p_h(p_h+1)^2} + \frac{-1}{p_h(p_h+1)}\right)\gamma^2|S_h|^2 \\ &= \left(\frac{4p_h^3+3p_h^2-p_h-1}{p_h(p_h+1)^3} + \frac{4p_h^2-p_h-3}{p_h(p_h+1)^2}\right)\gamma^2|S_h|^2. \end{aligned} \tag{21}$$

Since $p_h \ge 1$, it is easy to see from Eq. 21 that $f_2(X\cup\{x\}) - f_2(X) \ge 0$, i.e., f_2 increases monotonically. Hence, $-\mathbb{V}[L]$ is monotonically increasing in the number of partitions $p_h, \forall h \in H$. □

B.3 Submodularity (Theorem 4.7)

Proof. Similar to the proof of Lemma A.2, we prove by induction, and only need to show that for $Y = X \cup \{y\}$, where $y \notin X$, and $x = (h, 0, j) \notin X$ the following inequality holds.

$$(f_2(X\cup\{x\}) - f_2(X)) - (f_2(Y\cup\{x\}) - f_2(Y)) \ge 0. \tag{22}$$

Let $y = (b, 0, i)$ where $1 \le i \le |S_b|$, then there are two cases.
Case 1: $b \ne h$. So $(f_2(X\cup\{x\}) - f_2(X)) - (f_2(Y\cup\{x\}) - f_2(Y)) = 0$.

Case 2: $b = h$. Then

$$
\begin{aligned}
&(f_2(X \cup \{x\}) - f_2(X)) - (f_2(Y \cup \{x\}) - f_2(Y)) \\
=&(f_2(X \cup \{x\}) - f_2(X)) - (f_2(X \cup \{x, y\}) - f_2(X \cup \{x\})) \\
=&2 \cdot f_2(X \cup \{x\} - f_2(X) - f_2(X \cup \{x, y\} \\
=&2\left(2 - \frac{1}{p_h + 1}\right)\left(\gamma|S_h| + \frac{\alpha|S_h|^2}{p_h + 1}\right)^2 - \left(2 - \frac{1}{p_h}\right)\left(\gamma|S_h| + \alpha\frac{|S_h|^2}{p_h}\right)^2 \\
&\quad - \left(2 - \frac{1}{p_h + 2}\right)\left(\gamma|S_h| + \alpha\frac{|S_h|^2}{p_h + 2}\right)^2 \\
=&\frac{12p_h^5 + 48p_h^4 + 56p_h^3 + 6p_h^2 - 20p_h - 8}{p_h^3(p_h + 1)^3(p_h + 2)^3}\alpha^2|S_h|^4 \\
&+ \frac{8p_h^3 + 12p_h^2 - 8p_h - 8}{p_h^2(p_h + 1)^2(p_h + 2)^2}\gamma\alpha|S_h|^3 + \frac{-2}{p_h(p_h + 1)(p_h + 2)}\gamma^2|S_h|^2.
\end{aligned}
\tag{23}
$$

When $\alpha \geq \frac{p_h}{|S_h|}\gamma$, from Eq. 23 follows

$$
\begin{aligned}
&(f_2(X \cup \{x\}) - f_2(X)) - (f_2(Y \cup \{x\}) - f_2(Y)) \\
&\geq \frac{12p_h^5 + 48p_h^4 + 56p_h^3 + 6p_h^2 - 20p_h - 8}{p_h(p_h + 1)^3(p_h + 2)^3}\gamma^2|S_h|^2 \\
&\quad + \frac{8p_h^3 + 12p_h^2 - 8p_h - 8}{p_h(p_h + 1)^2(p_h + 2)^2}\gamma^2|S_h|^2 \\
&\quad + \frac{-2}{p_h(p_h + 1)(p_h + 2)}\gamma^2|S_h|^2 \\
&= \frac{12p_h^5 + 48p_h^4 + 56p_h^3 + 6p_h^2 - 20p_h - 8}{p_h(p_h + 1)^3(p_h + 2)^3}\gamma^2|S_h|^2 \\
&\quad + \frac{8p_h^3 + 10p_h^2 - 14p_h - 12}{p_h(p_h + 1)^2(p_h + 2)^2}\gamma^2|S_h|^2 \\
&= \frac{20p_h^5 + 82p_h^4 + 88p_h^3 - 28p_h^2 - 84p_h - 32}{p_h(p_h + 1)^3(p_h + 2)^3}\gamma^2|S_h|^2.
\end{aligned}
\tag{24}
$$

Clearly, for $\forall p_h \geq 1$, Eq. 24 > 0. Hence, the inequality (22) is satisfied, proving that f_2 is submodular when $\alpha \geq \frac{p_h}{|S_h|}\gamma$. □

C SCALING RESOURCES

In this set of experiments, we join the same data sets on clusters with various sizes. Specifically, we use Amazon's Elastic MapReduce (EMR) to set up clusters with 10, 20, 30, 40 and 50 workers of type m1.medium. As Fig. 6 shows, ExpVar beats the strongest competitor BKS+ on all cluster sizes. This shows that cluster size is not affecting the relative performance advantage of our partitioning method. We also observed that as the resources (number of workers) are scaled up, the running time improvement diminishes for all algorithms. The reason is that the increased communication cost in bigger clusters cannot be sufficiently compensated by the performance gain brought by parallelization. Eventually, the running times will not decrease as more workers are added, and might even start to increase at a certain point.

Note that choosing the right cluster size for a given data set is orthogonal to our approach. Some earlier work [16] has presented insights on how to achieve this. In this paper, we propose an algorithm that can decide a near-optimal partitioning solution for a given data set and a *given* number of workers.

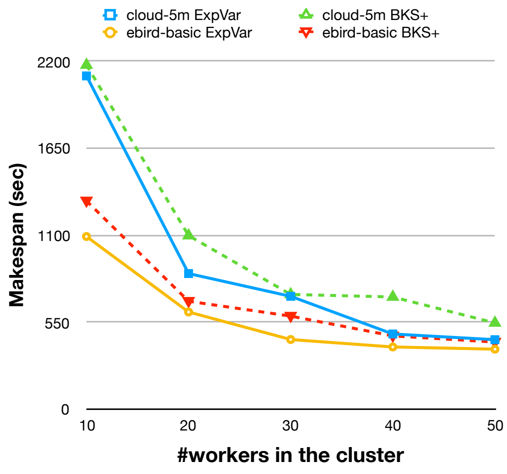

Figure 6: Running times of JOIN-AGG for cloud-5m and ebird-basic in clusters of different sizes. ExpVar and BKS+ are compared.

Data sets	ExpVar	NoPar	PaBr	PaBr+	BKS	BKS+
Zipf-90k-1.0-1g	1858	3529	3560	1950	2019	1933
Zipf-500k-1.0	3983	14509	7246	4057	7196	4136
cloud-200k	1965	7014	2871	1981	3401	2158

Table 9: Running times (sec) of JOIN-Fibo on Cluster14.

D RESULTS FOR JOIN-FIBONACCI

This set of experiments explores running time for JOIN-Fibo, which performs extra computation for each pair of joined tuples. In practice, this extra computation could be an algorithm computing a similarity score for two objects. Here, we use the Fibonacci computation as a tuneable stand-in for this post-processing cost. JOIN-Fibo executes the SQL query

```
SELECT S.*,T.*, Fibonacci(1000)
  FROM S JOIN T
    ON S.ID=T.ID
```

Due to the extra computation, the output-related cost increases even more compared to JOIN. This reduces the negative performance impact of over-partitioning (which affects only input cost), allowing more aggressive partitioning to improve load balancing. Table 9 lists the running times on Cluster14. As for JOIN, ExpVar, PaBr+ and BKS+ are comparable because they all avoid under-partitioning of large groups. In comparison, NoPar, PaBr and BKS suffer from under-partitioning and the resulting load imbalance.