# Processing Theta-Joins using MapReduce[*]

Alper Okcan
Northeastern University, Boston, MA
okcan@ccs.neu.edu

Mirek Riedewald
Northeastern University, Boston, MA
mirek@ccs.neu.edu

## ABSTRACT

Joins are essential for many data analysis tasks, but are not supported directly by the MapReduce paradigm. While there has been progress on equi-joins, implementation of join algorithms in MapReduce in general is not sufficiently understood. We study the problem of how to map arbitrary join conditions to Map and Reduce functions, i.e., a parallel infrastructure that controls data flow based on key-*equality* only. Our proposed join model simplifies creation of and reasoning about joins in MapReduce. Using this model, we derive a surprisingly simple randomized algorithm, called 1-Bucket-Theta, for implementing arbitrary joins (theta-joins) in a single MapReduce job. This algorithm only requires minimal statistics (input cardinality) and we provide evidence that for a variety of join problems, it is either close to optimal or the best possible option. For some of the problems where 1-Bucket-Theta is not the best choice, we show how to achieve better performance by exploiting additional input statistics. All algorithms can be made 'memory-aware', and they do not require any modifications to the MapReduce environment. Experiments show the effectiveness of our approach.

## Categories and Subject Descriptors

H.2.4 [**Systems**]: Distributed Databases; H.3.4 [**Systems and Software**]: Distributed Systems

## General Terms

Algorithms, Performance

## Keywords

MapReduce, Theta Join Processing, Skew

## 1. INTRODUCTION

Very large data sets pose a challenge in many disciplines. Internet companies want to analyze terabytes of application logs and clickstream data, scientists have to process data sets collected by large-scale experiments and sensors (e.g., Large Hadron Collider, National Virtual Observatory), and retailers want to find patterns in customer and sales data. When performing this analysis, parallel computation is essential to ensure reasonable *response time*. MapReduce [6] has emerged as probably the most popular paradigm for parallel processing, and it already has a great impact on data management research. One major reason for its success is the availability of a free open-source implementation, Hadoop [1], and an active developer community that keeps making improvements and adding features. Recently database-inspired high-level languages (PigLatin) [15] and support for SQL queries (Hive) [2] were added.

MapReduce is designed to process a single input data set, therefore joins are not directly supported. However, as recent research has shown, *equi*-joins can be implemented by exploiting MapReduce's key-equality based data flow management. But in many applications, more complex join predicates need to be supported as well. For spatial data, band-joins and spatial joins are common. Correlation analysis between data sets also requires similarity joins. And even traditional inequality predicates have received very little attention. The Map-Reduce-Merge extension [20] supports various join predicates, but it requires fundamental changes to MapReduce and how it is used. It not only adds a new Merge phase, but also requires the user to write code that explicitly has to be aware of the distributed nature of the implementation.

In this paper we propose techniques that enable efficient parallel execution of arbitrary theta-joins in MapReduce. No modifications of the MapReduce environment are necessary, and the user does not have to write any special-purpose code to manage data flow. Everything is achieved by simply specifying the appropriate (sequential) Map and Reduce functions. In particular, we make the following main contributions.

1. We propose a reducer-centered cost model and a join model that simplifies creation of and reasoning about possible theta-join implementations in MapReduce.

2. We propose a randomized algorithm called 1-Bucket-Theta for computing any theta-join, including the cross-product, in a *single* MapReduce job. This algorithm only needs minimal input statistics (cardinal-

[*]This work was supported by the National Science Foundation under Grant No. IIS-1017793. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*SIGMOD'11,* June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

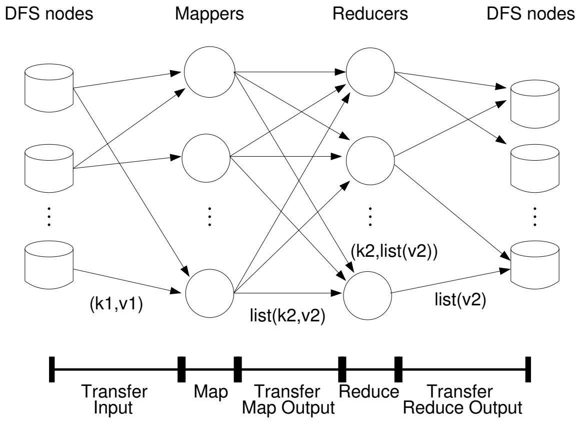


**Figure 1: MapReduce overview**

ity of input sets) and still effectively parallelizes any theta-join implementation. We show that it is close to optimal for joins with large output size. For highly selective joins, we show that even though better implementations in MapReduce might exist, they often cannot be used, leaving 1-Bucket-Theta as the best available option.

3. For a popular class of non-equi joins, including inequality and band-joins, we propose algorithms that often improve on 1-Bucket-Theta, as long as sufficiently detailed input statistics are available.

The rest of this paper is organized as follows. We survey MapReduce and a common equi-join implementation in Section 2. In Section 3, we present a qualitative MapReduce cost model and a join model. Using this join model, we derive the 1-Bucket-Theta algorithm and prove its properties in Section 4. Improvements for certain join types are discussed in Section 5. Section 6 shows representative experimental results, Section 7 discusses related work, and Section 8 concludes the paper.

# 2. MAPREDUCE AND JOINS

## 2.1 Overview of MapReduce

MapReduce was proposed to simplify large-scale data processing on distributed and parallel architectures, particularly clusters of commodity hardware [6]. The main idea of this programming model is to hide details of data distribution and load balancing and let the user focus on the data processing aspects. A MapReduce program consists of two primitives, Map and Reduce. The Map function is applied to an individual input record in order to compute a set of intermediate key/value pairs. For each key, Reduce works on the list of *all* values with this key. Any output produced by Reduce is written to a file stored in a distributed file system.

An overview of the MapReduce architecture is given in Figure 1. Input records might be distributed across several physical locations on a distributed file system (DFS). Once the MapReduce job is initialized, these records are transferred to mapper nodes in chunks. For each input record, a new instance of Map is executed. It parses the record as a key/value pair of type $(k_1, v_1)$ and outputs new key/value pairs of type $(k_2, v_2)$. These Map outputs are collected locally on the mapper node. Based on a (default or user-defined) function, the keys of type $k_2$ are assigned to reducer nodes. Mapper nodes shuffle their intermediate output to create a list of key/value pairs for each reducer node. These lists are transferred to the appropriate reducer nodes. Once the map phase is completed, each reducer node sorts its input by key. Each resulting pair of type $(k_2, \text{list}(v_2))$ is then processed by an invocation of the Reduce function. Reduce's output, of type $\text{list}(v_2)$ according to the original MapReduce paper, is then transferred to DFS nodes. Notice that Reduce could also produce output of a different type $\text{list}(v_3)$.

## 2.2 Example: Equi-Join

Consider an equi-join of data sets $S$ and $T$ on a common attribute $A$, i.e., join condition $S.A = T.A$. Commonly this is implemented by making the join attribute the key, ensuring that all tuples with identical join attribute values are processed together in a single invocation of the Reduce function. More precisely, for each input tuple $s \in S$, Map outputs the key-value pair $(s.A, s)$. Notice that $s$ is also augmented by adding an attribute `origin` which indicates that the tuple came from $S$. Tuples from $T$ are processed similarly. For each join attribute value, Reduce then computes the cross-product between the corresponding tuples that have origin $S$ with the tuples whose origin is $T$.

This is the implementation commonly seen in the literature (see Section 7), we therefore refer to it as the **Standard Equi-Join Implementation**.

This algorithm suffers from two problems. First, the number of reducer nodes is limited by the number of distinct values of $A$ in the input data sets. The reason is that all tuples with the same $A$-value have to be processed by the same invocation of Reduce, thus inherently limiting parallelism. The second problem is caused by data skew. If some $A$-value occurs very frequently, the reducer processing it receives an overly large share of work, both for processing the input and writing the large result to the DFS, thus delaying the completion of the job.

## 2.3 Example: Inequality-Join

Consider a join between data sets $S$ and $T$ with an inequality condition like $S.A \leq T.A$. Such joins seem inherently difficult for MapReduce, because each $T$-tuple has to be joined not only with $S$-tuples that have the *same* $A$ value, but also those with *different* (smaller) $A$ values. It is not obvious how to map the inequality join condition to a key-equality based computing paradigm.

One might consider the following "naive" approach. Assume all values of $A$ are non-negative integers. To ensure that a $T$-tuple joins with all $S$-tuples of equal or smaller $A$-value, we can make Map output each $T$-tuple for all possible smaller $A$-values as keys. More precisely, for each input tuple $s \in S$, Map only outputs $(s.A, s)$, but for each $t \in T$, it outputs $(a, t)$ for *every* $a \leq T.A$.

However, this algorithm also suffers from two major problems. First, it generates a potentially huge amount of "duplicates" of $T$-tuples that depends on the values of the join attribute. Second, if attribute $A$ is not integer or can have negative values, one cannot enumerate all smaller values of $A$ for a given value $t.A$. For these cases, the Map function needs to know the set of distinct values of $A$ in $S$ to produce the right duplicates of $T$.

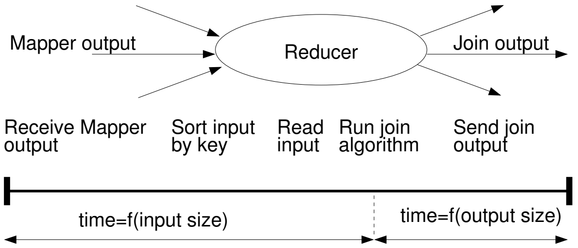


**Figure 2: Processing pipeline at a reducer**

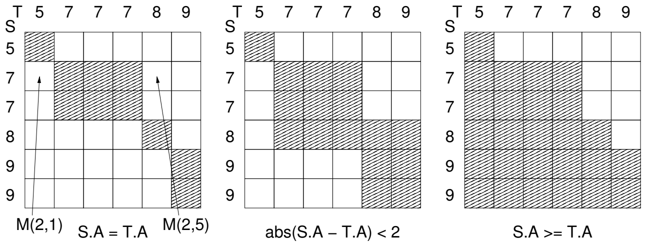


**Figure 3: Join matrices for equi-join, similarity-join, and inequality-join. Numbers indicate join attribute values from $S$ and $T$, shaded cells indicate join results. $M(i,j)$ indicates cell numbering.**

# 3. PRELIMINARIES

In the following analysis, we generally assume that all reducer nodes by design have approximately the same computational capabilities. This holds in practice for clusters of virtual machines created in the Cloud, but also for physical clusters running on commodity hardware.

## 3.1 Optimization Goal

For a given join operator and its inputs, we want to **minimize job completion time** for a given number of processing nodes. Job completion time includes all phases of MapReduce, from when the first data tuple is transferred to a mapper node until the last output tuple is written back to the DFS. Short job completion time is desirable from a user's point of view. As we discuss below, it also inherently leads to a load-balancing approach. This is in line with previous work on distributed and parallel systems, where load balancing ideas play a central role.

## 3.2 Cost Model for MapReduce

Competing MapReduce implementations for a given join problem can only differ in their Map and Reduce functions. Since the cost for transferring data from the DFS to the mapper nodes and the cost for reading the input tuples locally at each mapper is not affected by the concrete Map and Reduce functions, we do not need to take these costs into account for the optimization. Map and Reduce functions affect the costs from producing Map function output until writing the final join result back to the DFS. To analyze the completion time of these MapReduce job phases, consider a single reducer. The reducer receives a subset of the mapper output tuples as its input. It sorts the input by key, reads the corresponding value-list for a key, computes the join for this list, and then writes its locally created join tuples to the DFS. Figure 2 illustrates this process.

Due to the nature of MapReduce, it is easy to balance load between mapper nodes. On the other hand, when using the standard equi-join algorithm, it is possible for some reducer to receive a much larger share of work, which delays job completion. To minimize job completion time, we therefore need to minimize the greatest amount of work that is assigned to any reducer, i.e., *balance load* between reducers as evenly as possible.

As Figure 2 illustrates, some of the reducer's time is spent on tasks whose duration depends on input size, while others depend on output size or a combination of both. In general costs are monotonic, i.e., the greater the input size, the greater the time spent processing it (similarly for output size). We can therefore evaluate the quality of a join algorithm by its **max-reducer-input** and its **max-reducer-output**, i.e., the maximum over the input sizes assigned to any reducer and output sizes produced by any reducer, respectively.

We distinguish between the following cases. We say that a join problem is **input-size dominated** if reducer-input related costs dominate job completion time. If reducer-output related costs dominate job completion time, then the join problem is **output-size dominated**. If neither clearly dominates the other, we have an **input-output balanced** problem. Notice that the join problem category depends on the specific join implementation selected. For input-size dominated problems, job completion time is minimized by minimizing max-reducer-input. For output-size dominated problems, we need to minimize max-reducer-output. And for input-output balanced problems, we have to minimize a combination of both.

Notice that our qualitative cost model makes no assumption about which type of cost dominates, i.e., it includes cases where network transfer time, CPU-time, or local I/O-time dominate. All these costs tend to increase with increasing input and/or output size, hence minimizing the maximal input and/or output size minimizes job completion time, no matter if network, CPU, or local I/O is the bottleneck.

## 3.3 Theta Join Model

We model a join between two data sets $S$ and $T$ with a **join-matrix** $M$ and employ this representation for creation of and reasoning about different join implementations in MapReduce. Figure 3 shows example data sets and the corresponding matrix for a variety of join predicates. For row $i$ and column $j$, matrix entry $M(i,j)$ is set to `true` (shaded in the picture) if the $i$-th tuple from $S$ and $j$-th tuple from $T$ satisfy the join condition, and `false` (not filled) otherwise. Since any theta-join is a subset of the cross-product, the matrix can represent any join condition.

## 3.4 Mapping Join Matrix Cells to Reducers

Our goal is to have each join output tuple be produced by exactly one reducer, so that expensive post-processing or duplicate elimination is avoided. Hence, given $r$ reducers we want to map each matrix cell with value $M(i,j) =$ `true` to exactly one of the $r$ reducers. We will also say that reducer $R$ *covers* a join matrix cell, if this cell is mapped to $R$.

There are many possible mappings that cover all `true`-valued matrix cells. **Our goal is to find that mapping from join matrix cells to reducers that minimizes job completion time.** Hence we want to find mappings that either balance reducer input share (for input-size dominated

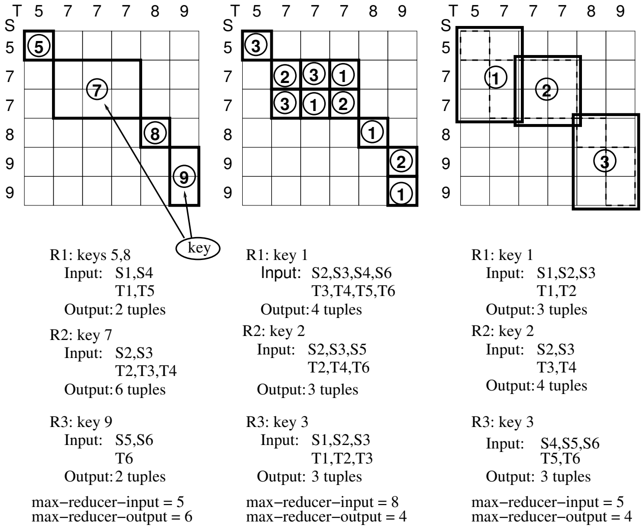


**Figure 4: Matrix-to-reducer mappings for standard equi-join algorithm (left), random (center), and balanced (right) approach**

joins), or balance reducer output share (for output-size dominated joins), or achieve a compromise between both (for input-output balanced joins).

Figure 4 illustrates the tradeoffs in choosing different mappings. The left image is the mapping used by the standard equi-join implementation in MapReduce. All tuples with the same join attribute value are mapped to the same reducer. This results in a poor balance of both reducer input and output load. E.g., reducer R2 receives 5 input tuples and creates 6 output tuples, while reducer R3 works with 3 input and 2 output tuples.

The other two images correspond to new equi-join algorithms that we have not seen in the literature before. (Using our formulation of theta-join implementations as a mapping from `true` matrix entries to the set of reducers, it is easy to come up with many more algorithms.) The center image represents a very fine-grained mapping. Even though the 5-th and 6-th tuple from $S$ (S5, S6) and the 6-th tuple from $T$ (T6) all have the same join attribute value, result tuple (S5,T6) is produced by reducer R2, while (S6,T6) is produced by reducer R1. The example also illustrates how skew is effectively addressed, e.g., by breaking the big output chunk for tuples with join value 7 into many small pieces. The downside of the better output load balancing is the significantly greater input size for every reducer, caused by the duplication of tuples to enable each reducer to generate the desired results. E.g., the second and third tuples from $S$ have to be sent to all three reducers. Also notice that both R2 and R3 could produce outputs (S2,T2) and (S3,T2), because they both have the corresponding input tuples. To enforce the matrix-to-reducer mapping (and avoid duplicate output), the algorithm would have to pass information about the mapping to each reducer.

The mapping on the right illustrates how we can achieve the best of both worlds. Overly large output chunks are effectively broken up, while input duplication is kept low and reducer input and output are both well-balanced. This is achieved despite the mapping covering not only `true` cells, but also some like $M(2,1)$ that do not contribute to the join output. (Those do not affect the join result, because Reduce eliminates them.) Our new algorithms represent practical implementations of this basic idea: balance input and output costs while minimizing duplication of reducer input tuples. We will repeatedly make use of the following important lemma.

LEMMA 1. *A reducer that is assigned to c cells of the join matrix $M$ will receive at least $2\sqrt{c}$ input tuples.*

PROOF. Consider a reducer that receives $m$ tuples from $S$ and $n$ tuples from $T$. This reducer can cover at most $m \cdot n$ cells of the join matrix $M$. Hence to cover $c$ matrix cells, it has to hold that $m \cdot n \geq c$. Considering all possible non-negative values $m$ and $n$ that satisfy $m \cdot n \geq c$, the sum of $m$ and $n$ is minimized for $m = n = \sqrt{c}$. □

## 4. THE 1-BUCKET-THETA ALGORITHM

The examples in Section 2 illustrate the challenges for implementing joins in MapReduce: data skew and the difficulty of implementing non-equi-joins with key-equality based data flow control. We now introduce 1-Bucket-Theta, an algorithm that addresses these challenges, and provide strong analytical results about its properties.

### 4.1 Implementing the Cross-Product

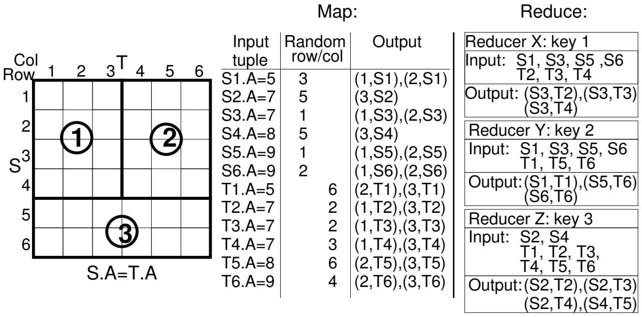
Since the cross-product combines every tuple from $S$ with every tuple from $T$, the corresponding join matrix has all entries set to `true`. We explain how 1-Bucket-Theta performs matrix-to-reducer mapping, show that it is near-optimal for computing the cross-product, and discuss how these results extend to processing of theta-joins.

#### *4.1.1 Analytical Results*

We first consider balancing output-related costs across reducers. Since there are $|S||T|$ output tuples to be produced by $r$ reducers, the lower bound for max-reducer-output is $|S||T|/r$. (As usual, $|S|$ denotes the cardinality of a set $S$.) Together with Lemma 1, this implies a lower bound of $2\sqrt{|S||T|/r}$ for max-reducer-input, giving us the following lemma:

LEMMA 2. *For any matrix-to-reducer mapping for the cross-product $S \times T$, $|S||T|/r$ and $2\sqrt{|S||T|/r}$ are the lower bounds for max-reducer-output and max-reducer-input, respectively.*

To match the lower bound for max-reducer-output, the matrix-to-reducer mapping has to partition the matrix such that exactly $|S||T|/r$ of the matrix cells are mapped to each of the $r$ reducers. Notice that if cell $M(i,j)$ is assigned to reducer $k$, then reducer $k$ needs to have both the $i$-th tuple from $S$ and the $j$-th tuple from $T$ to be able to create the combined output tuple. Hence the $i$-th tuple from $S$ has to be sent to each reducer whose region in the matrix intersects the $i$-th row. Similarly, the $j$-th tuple from $T$ has to be sent to all reducers whose regions intersect the $j$-th column of $M$. As Figure 4 illustrates, depending on the number of different reducers assigned to cells in each row and column, input tuples might be duplicated many times. As the following theorem shows, for some special cases we can actually match both lower bounds with a square-based matrix-to-reducer mapping.



Map:

| Input tuple | Random row/col | Output |
|---|---|---|
| S1.A=5 | 3 | (1,S1),(2,S1) |
| S2.A=7 | 5 | (3,S2) |
| S3.A=7 | 1 | (1,S3),(2,S3) |
| S4.A=8 | 5 | (3,S4) |
| S5.A=9 | 1 | (1,S5),(2,S5) |
| S6.A=9 | 2 | (1,S6),(2,S6) |
| T1.A=5 | 6 | (2,T1),(3,T1) |
| T2.A=7 | 2 | (1,T2),(3,T2) |
| T3.A=7 | 2 | (1,T3),(3,T3) |
| T4.A=7 | 3 | (1,T4),(3,T4) |
| T5.A=8 | 6 | (2,T5),(3,T5) |
| T6.A=9 | 4 | (2,T6),(3,T6) |

Reduce:

Reducer X: key 1
Input: S1, S3, S5 ,S6
T2, T3, T4
Output: (S3,T2),(S3,T3)
(S3,T4)

Reducer Y: key 2
Input: S1, S3, S5, S6
T1, T5, T6
Output:(S1,T1),(S5,T6)
(S6,T6)

Reducer Z: key 3
Input: S2, S4
T1, T2, T3,
T4, T5, T6
Output:(S2,T2),(S2,T3)
(S2,T4),(S4,T5)

**Figure 5: Matrix-to-reducer mapping for equi-join. The join matrix is partitioned into 3 regions, each assigned to one reducer.**

THEOREM 1. *For cross-product $S \times T$, assume $|S|$ and $|T|$ are multiples of $\sqrt{|S||T|/r}$, i.e., $|S| = c_s\sqrt{|S||T|/r}$ and $|T| = c_T\sqrt{|S||T|/r}$ for integers $c_S, c_T > 0$. Under these conditions, the join matrix $M$ can be partitioned into $c_S$ by $c_T$ squares of size $\sqrt{|S||T|/r}$ by $\sqrt{|S||T|/r}$ each. The corresponding matrix-to-reducer mapping matches the lower bounds for both max-reducer-output and max-reducer-input.*

For other examples where $|S|$, $|T|$, and $r$ do not satisfy the properties required by Theorem 1, the problem of minimizing max-reducer-input for a given value of max-reducer-output can be formulated as an integer linear programming problem. These problems are generally NP-hard, hence it can be expensive to solve. However, we can show that we can always find a solution that is "close" to optimal at low cost.

Without loss of generality, let $|S| \leq |T|$. We first consider an extreme case where $|S|$ is much smaller than $|T|$, more precisely: $|S| < |T|/r$. This implies $|S| < \sqrt{|S||T|/r}$, i.e., the side length of the optimal square that matches the lower bounds is "taller" than the join matrix. Hence the lower bounds are not tight, because no partition of the matrix can have more than $|S|$ tuples from input set $S$. It is easy to see that the optimal partitioning of the matrix into $r$ regions would then consist of rectangles of size $|S|$ by $|T|/r$.

THEOREM 2. *For $S \times T$, consider matrix-to-reducer mappings that perfectly balance the entire output to $|S||T|/r$ tuples per reducer. Let $|S| < |T|/r$. Under this condition, max-reducer-input is minimized by partitioning the matrix into a single row of $r$ rectangles of size $|S|$ by $|T|/r$.*

Now consider the remaining case, i.e., $|T|/r \leq |S| \leq |T|$. Let $c_S = \lfloor |S|/\sqrt{|S||T|/r} \rfloor$ and $c_T = \lfloor |T|/\sqrt{|S||T|/r} \rfloor$. Intuitively, $c_S$ and $c_T$ indicate how many optimal squares of side-length $\sqrt{|S||T|/r}$ we can fit into the matrix horizontally and vertically, respectively. From $|T|/r \leq |S| \leq |T|$ follows that $c_S \geq 1$ and $c_T \geq 1$. After creating these squares, starting in the upper left corner of the matrix, there might be some cells that are not covered. These cells are in a horizontal "stripe" whose height is less than $\sqrt{|S||T|/r}$ (otherwise another row of optimal squares would have fit there) and a vertical "stripe" whose width is less than $\sqrt{|S||T|/r}$ (otherwise another column of optimal squares would have fit there). We can cover both stripes by increasing each region's height and width by a factor of $(1 + 1/\min\{c_S, c_T\})$. Since both $c_S, c_T \geq 1$, this factor is at most 2.

This implies that we can cover the entire matrix with $c_S$ rows, each consisting of $c_T$ regions that are squares of side-length $(1 + 1/\min\{c_S, c_T\})\sqrt{|S||T|/r} \leq 2\sqrt{|S||T|/r}$. Hence no reducer produces more than $4|S||T|/r$ output tuples and no reducer receives more than $4\sqrt{|S||T|/r}$ input tuples. The following theorem summarizes this result.

THEOREM 3. *For $S \times T$, let $|T|/r \leq |S| \leq |T|$. Under this condition, we can always find a matrix-to-reducer mapping with the following properties. (1) No reducer produces more than $4|S||T|/r$ output tuples, i.e., at most 4 times the lower bound of $|S||T|/r$ for max-reducer-output. And (2) no reducer receives more than $4\sqrt{|S||T|/r}$ input tuples, i.e., at most 2 times the lower bound of $2\sqrt{|S||T|/r}$ for max-reducer-input.*

Together, Theorems 1, 2, and 3 give us strong guarantees for the near-optimality of our matrix-to-reducer mappings for implementing the cross product. We can *always* find a solution where none of the reducers receives more than twice its "fair share" of the input-related cost and four times its "fair share" of the output-related cost. In practice this bound is usually much better. For instance, if $c_S$ and $c_T$ as introduced above are in the order of 10, then our scheme guarantees that no reducer receives more than 1.1 times its fair input share and 1.21 times its fair output share.

### 4.1.2 From Mapping to Randomized Algorithm

Turning a matrix-to-reducer mapping into a MapReduce algorithm is conceptually straightforward. For an incoming $S$-tuple, the Map function finds all regions intersecting the row corresponding to that tuple in the matrix. For each region, it creates an output pair consisting of the region key and the tuple (augmented by the `origin` attribute as described earlier). Similarly, for each $T$-tuple, the Map function finds the partitions intersecting the corresponding column in the matrix, etc.

Consider the partitions in Figure 5 for computing the equi-join (using the cross-product of the data sets) for the same data sets as in Figure 4. Ideally we would like to assign the $i$-th $S$-tuple to row $i$ in the matrix, similarly for $T$. Since all regions have the same size (12 cells), this would guarantee that each reducer is responsible for its exact fair share of $12/36 = 1/3$ of the cross-product result. The problem is for Map to know which row in the matrix an incoming $S$-tuple belongs to. Each invocation of Map *by design* sees only a single $S$ or $T$ tuple. There is no consensus mechanism for a Map instance to find out how many $S$ values in the data set are smaller than the one it is currently processing. This implies that there is no *deterministic* way for Map to make that decision. To guarantee that tuples are assigned to the appropriate rows and columns, Map would need to know that assignment. This could be achieved by running another MapReduce pre-processing step, assigning unique row and column numbers to each $S$ and $T$ tuple.

We can avoid this overhead by making Map a randomized algorithm as shown in Algorithm 1. It can implement any theta-join of two input sets $S$ and $T$. For each incoming $S$-tuple, Map randomly chooses a row in matrix $M$. It then creates an output tuple for each region that intersects with this row. Note that the matrix-to-reducer mapping is computed before this MapReduce job starts. In the example shown in Figure 5, the first $S$-tuple is mapped to row 3.

Since row 3 intersects with regions 1 and 2, a corresponding Map output tuple is created for both. Input tuples from $T$ are processed similarly, by choosing a random *column* in matrix $M$. Notice how there are some rows and columns that are randomly selected for multiple input tuples, while others are not selected at all. While this does not *guarantee* the desired input and output size for each reducer any more, the large data size makes significant variations extremely unlikely. Randomization also turns out to be a crucial feature for theta-joins that are not cross-products (see Section 4.2 and experiments).

Consider a reducer node $R$ that should receive $n_R$ of the $|S|$ tuples from $S$ based on the matrix-to-reducer mapping. E.g., in Figure 5 reducer 1 should receive 4 of the 6 $S$-tuples. Let $X_1$, $X_2$,..., $X_{|S|}$ be random variables, such that $X_i = 1$ if the $i$-th tuple from $S$ is assigned to this reducer $R$, and 0 otherwise. Since $S$-tuples are assigned to rows uniformly at random, we obtain $p_i = \Pr[X_i = 1] = n_R/|S|$. The $X_i$ are independent random variables, therefore the Chernoff Bound gives us

$$\Pr[X \geq (1+\delta)\mu] \leq \mathrm{e}^{-\mu((1+\delta)ln(1+\delta)-\delta)}.$$

Here $X = \sum_i X_i$, i.e., $X$ is the number of $S$-tuples assigned to reducer $R$. And $\mu = \mathbb{E}[X]$ is the expected number of tuples assigned to that reducer.

In practice this bound is very tight as long as the reducer with the greatest input share receives at least around 10,000 tuples. This is usually the case in practice when dealing with large data sets. For example, let $|S| = 10,000,000$. If reducer $R$ is supposed to receive 10,000 out of 10,000,000 $S$-tuples, the probability that it will receive an extra 10% or more input tuples is $10^{-22}$, i.e., virtually zero. Since it is almost impossible for a reducer to receive more than 110% of its target input, its output is practically guaranteed to not exceed $1.1^2 = 1.21$ times its target size. Our experiments support this result.

The alert reader will have realized that considering the cross-product does not seem like the most efficient algorithm for the equi-join example in Figure 5. In fact, the three regions in the right example in Figure 4 provide a superior partitioning that avoids covering matrix cells that are not part of the join result. We explore this issue in more depth in Sections 4.2 and 5.

## 4.2 Implementing Theta-Joins

Algorithms 1 and 2 can implement any theta-join by selecting the appropriate MyFavoriteJoinAlg for Reduce. While we showed above that this MapReduce implementation is close to optimal for cross-product computation, this result does not necessarily carry over to arbitrary joins. **This section provides strong evidence that even for very selective join conditions, it is often not possible to find a better algorithm than 1-Bucket-Theta.** This does not mean that such better algorithms do not exist. They just cannot be identified as correct implementations with the information available at the time when the best implementation is selected for a given join problem, as we show now.

Consider an arbitrary theta-join with selectivity $\sigma$, i.e., it produces $\sigma|S||T|$ output tuples. To minimize max-reducer-output, each reducer should be responsible for $\sigma|S||T|/r$ join output tuples. While 1-Bucket-Theta practically guarantees to balance the *cross-product* output across reducers, this might not be true for other joins. For example, on some reducer almost all cross-product tuples might satisfy the join condition, while almost none do so on another. Fortunately this is very unlikely because of the randomization that assigns random samples from $S$ and $T$ to each reducer. While we do not have an analytical proof, our experiments show that join output is generally very evenly distributed over the reducers. This is to be expected as long as join output size is large enough so that sampling variance is "averaged out". Significant variance in output size is only likely when join output size is very small, e.g., below thousands of tuples per reducer. However for those cases the *total* join output size is so small that even a significant output imbalance has only a small effect on the *absolute* runtime. (Note that MapReduce typically works with data chunks of size 64 megabytes or larger.)

**Algorithm 1** : Map (Theta-Join)

```
Input: input tuple $x \in S \cup T$
  /* matrix to regionID mapping is loaded into a lookup table during initialization of mapper */
if $x \in S$ then
  matrixRow = random(1,$|S|$)
  for all regionID in lookup.getRegions(matrixRow) do
    Output (regionID, ($x$, "S") ) /* key: regionID */
else
  matrixCol = random(1,$|T|$)
  for all regionID in lookup.getRegions(matrixCol) do
    Output (regionID, ($x$, "T") )
```

**Algorithm 2** : Reduce (Theta-Join)

```
Input: (ID, $[(x_1, \text{origin}_1), (x_2, \text{origin}_2), \ldots, (x_k, \text{origin}_k)]$)
Stuples = $\emptyset$; Ttuples = $\emptyset$
for all $(x_i, \text{origin}_i)$ in input list do
  if $\text{origin}_i$ = "S" then
    Stuples = Stuples $\cup \{x_i\}$
  else
    Ttuples = Ttuples $\cup \{x_i\}$
joinResult = MyFavoriteJoinAlg(Stuples, Ttuples)
Output( joinResult )
```

In short, whenever join output size is large enough to significantly affect job completion time, 1-Bucket-Theta's randomized approach balances output very well across reducers. It therefore is very difficult to beat it on output-related costs. For another algorithm to achieve significantly lower total job completion time, it has to have significantly lower *input*-related costs than 1-Bucket-Theta.

LEMMA 3. *Let $1 \geq x > 0$. Any matrix-to-reducer mapping that has to cover at least $x|S||T|$ of the $|S||T|$ cells of the join matrix, has a max-reducer-input value of at least $2\sqrt{x|S||T|/r}$.*

PROOF. If $x|S||T|$ cells of the matrix have to be covered by $r$ regions (one for each reducer), it follows from the pigeonhole principle that at least one reducer has to cover $x|S||T|/r$ or more cells. This together with Lemma 1 implies that at least $2\sqrt{x|S||T|/r}$ input tuples need to be sent to that reducer. □

As we showed in Section 4.1.1, 1-Bucket-Theta virtually guarantees that its max-reducer-input value is at most

$4\sqrt{|S||T|/r}$, and usually it is much closer to $2\sqrt{|S||T|/r}$. Hence the ratio between max-reducer-input of 1-Bucket-Theta versus any competing theta-join algorithm using a different matrix-to-reducer mapping is at most

$$\frac{4\sqrt{|S||T|/r}}{2\sqrt{x|S||T|/r}} = \frac{2}{\sqrt{x}}.$$

E.g., compared to any competing join implementation whose matrix-to-reducer mapping has to cover 50% or more of the join matrix cells, 1-Bucket-Theta's max-reducer-input is at most about 3 times the max-reducer-input of that algorithm. Notice that this is an upper bound that is quite loose in practice. E.g., when working with 100 reducer nodes and inputs that are of similar sizes, e.g., where one is at most 4 times larger than the other, max-reducer-input is closer to $2.5\sqrt{|S||T|/r}$ (instead of $4\sqrt{|S||T|/r}$). Then the worst-case ratio for any mapping covering at least 50% of the join matrix is only $1.25/\sqrt{0.5} \approx 1.8$.

In summary, unless $x$ is very small, no other matrix-to-reducer mapping is going to result in significantly lower, e.g., by a factor of more than 3, max-reducer-input compared to using 1-Bucket-Theta. Stated differently, **the only way to significantly improve over the job completion time of 1-Bucket-Theta is to find a matrix-to-reducer mapping that does not assign a significant percentage of the join matrix cells to any reducer, e.g., at least 50%.**

Recall that for correctness, every join matrix cell with value `true` has to be assigned to a reducer (see Section 3.4). This means that for any join that produces a large fraction of the cross-product, 1-Bucket-Theta is also guaranteed to be close to optimal in terms of both max-reducer-input and max-reducer-output.

For joins with very selective conditions, usually a matrix-to-reducer mapping will exist that has a significantly lower max-reducer-input than 1-Bucket-Theta. To improve over 1-Bucket-Theta, we have to *find* such a mapping. Because of Lemma 3, a necessary condition for this mapping is that it covers only a relatively small percentage of the join matrix cells, e.g., less than 50%. As the following discussion shows, it is often difficult in practice to find a mapping with this property due to requirements for both input statistics and join condition.

**Input statistics.** Knowing only the cardinality of $S$ and $T$, it is not possible to decide for any matrix cell if it is part of the join output or not. Let $(s,t)$ be a pair of an $S$ and a $T$ tuple that satisfies the join condition. If the join algorithm assumes that some matrix cell $M(i,j)$ is not part of the join result, one can easily construct a counter example by creating sets $S$ and $T$ where $s$ and $t$ are assigned to the $i$-th row and $j$-th column of $M$, respectively. To identify matrix cells that do not need to be covered, more detailed input statistics are required.

**Join condition.** If the join condition is a user-defined blackbox function, then we do not know which join matrix cells have value `false` unless we actually evaluate the join condition for these cells. However, this defeats the purpose of the algorithm: To find an efficient join algorithm, we would actually have to compute the join for all cells we are considering as candidates for *not* covering them by any reducer. Even if a join condition does not contain user-defined functions, in practice it is often difficult to identify large regions in the join matrix for which the algorithm can be certain that the entire region contains no join result tuple.

Before we explore this issue in more depth, consider the right example in Figure 4 and the example in Figure 5. In both cases we compute the same equi-join for the same inputs. The partitioning in the right example in Figure 4 is better, because it avoids the high input duplication needed for 1-Bucket-Theta's cross-product based computation. This is achieved by not covering large regions of cells that contain no result tuples (lower-left and upper-right corners).

For a set $C$ of matrix cells of interest, input statistics give us predicates that hold for this set. Consider the right example in Figure 4. There the $S$-tuple with the $i$-th largest value of the join attribute is mapped to row $i$ (similarly for $T$-tuples and matrix columns). For the block of 3 by 4 cells in the lower-left corner, histograms on $S$ and $T$ could imply that predicate $(S.A \geq 8 \wedge T.A \leq 7)$ holds for this matrix region. To be able to not assign any cell in this block to a reducer, the algorithm has to know that none of the cells in the region satisfies the join condition. In the example, it has to show that $\forall s \in S, t \in T : (s.A \geq 8 \wedge t.A \leq 7) \Rightarrow \neg(s.A = t.A)$. While this is straightforward for an equi-join, it can be difficult and expensive in general.

To summarize our results: For selective join conditions, better algorithms than 1-Bucket-Theta might exist. These algorithms require that a significant fraction of the join matrix cells not be assigned to any reducer. Unfortunately, in practice it can be impossible (insufficient input statistics, user-defined join conditions, complex join conditions) or computationally very expensive to find enough of such matrix cells. For those cases, even if we could guess a better matrix-to-reducer mapping, we could not use it because there is no proof that it does not miss any output tuple. The next section will explore special cases where the join condition admits an efficient algorithm for identifying regions in the join matrix that do not contain any join result tuples.

## 5. EXPLOITING STATISTICS

We present algorithms that improve over 1-Bucket-Theta for popular join types.

### 5.1 Approximate Equi-Depth Histograms

1-Bucket-Theta only needed to know the cardinality of the inputs. To improve over it, we showed in the previous section that we need to identify large regions in the join matrix that do not contain any output tuples. This requires more detailed input statistics, which have to be computed on-the-fly if they are not available.

We can compute an approximate equi-depth histogram on input sets $S$ and $T$ with two scans as follows. In the first pass, we sample approximately $n$ records from each $S$ and $T$ in a MapReduce job. For an input tuple from $S$, Map decides with probability $n/|S|$ to output the tuple, otherwise discards it. These $n$ records are sorted by join attribute and grouped in a single reduce task to compute approximate $k$-quantiles, $k < n$, that are used as the histogram's bucket boundaries. ($T$ is processed analogously.) In a second MapReduce job, we make another pass on both data sets and count the number of records that fall into each bucket. For join types where it is beneficial to have histogram bucket boundaries between $S$ and $T$ line up, we perform the second

```
Algorithm 3 : M-Bucket-I
Input: maxInput, r, M
row = 0
while row < M.noOfRows do
    (row, r) = CoverSubMatrix(row, maxInput, r, M)
    if r < 0 then
        return false
return true
```

phases for $S$ and $T$ together and use the union of both sets' bucket boundaries for the final histogram for each set.

Using such a histogram, it is straightforward to identify 'empty' regions in the join matrix for a popular class of joins including equi-joins , band-joins, and inequality-joins. As discussed earlier, only non-empty matrix regions, i.e., those containing at least one result record, need to be assigned to a reducer. We refer to cells in those regions as **candidate cells**.

## 5.2 M-Bucket-I and M-Bucket-O

For input-size dominated joins, we want to find a matrix-to-reducer mapping that minimizes max-reducer-input. Finding such an optimal cover of all candidate cells in general is a hard problem, hence we propose a fast heuristic. We refer to the corresponding algorithm as M-Bucket-I, because it needs more detailed input statistics (**M**ultiple-bucket histogram) and minimizes max-reducer-**I**nput.

The pseudo-code of M-Bucket-I is shown in Algorithm 3. Given the desired number of regions ($r$), the maximal input size allowed for each region (*maxInput*), and a join matrix $M$, the algorithm divides the covering problem into sub-problems of covering sub-regions of the join matrix at each step. In order to preserve similar sub-problems, M-Bucket-I only considers horizontal fragments of the matrix. Starting from the top row, it tries to cover all candidate cells in a block of consecutive rows. Then it repeats the same process starting at the next row that is not covered yet. It continues to cover blocks of consecutive rows until it has either covered all candidate cells, or it exhausted the $r$ regions without being able to cover all candidate cells.

During each execution of the while-loop, M-Bucket-I covers a block of rows as shown in Algorithm 4. The number of rows in a block can vary between different calls of Algorithm 4. For all blocks starting at row $\text{row}_s$ and consisting of $i$ rows, $i \in [1, maxInput - 1]$, Algorithm 4 computes a score. The score is defined as the average number of candidate cells covered by each region in the block. (Intuitively, we want to cover as many candidate cells with as few regions as possible.) Among the possible $maxInput - 1$ blocks starting at $\text{row}_s$, the block with the highest score is selected. Now all candidate cells in this block are covered, and the process continues in the next row right below the block. Our algorithm does not consider blocks of more than $maxInput - 1$ rows in order to reduce the search space (see for-loop limits). This works very well in practice, because 'taller' blocks usually result in tall slim regions that have a low score.

Given the first row $\text{row}_f$ and the last row $\text{row}_l$ of a particular block of rows in $M$, M-Bucket-I assigns candidate cells within the block *column-by-column* as shown in Algorithm 5. It starts by creating a new region with an initial input capacity of *maxInput*. M-Bucket-I iterates through each column $c_i$ and assigns all candidate cells within $c_i$ (between $\text{row}_f$ and $\text{row}_l$) to the region as long as the input capacity is not exceeded. When adding candidate cells in a column $c_i$ would result in the region exceeding its input limit, a new region is created and $c_i$ and the next columns are assigned to that region, until it reaches the input size limit, and so on. Once all columns of the block are covered, Algorithm 5 returns the set of cover regions it created.

```
Algorithm 4 : CoverSubMatrix (M-Bucket-I)
Input: $\text{row}_s$, maxInput, r, M
maxScore = -1, rUsed = 0
for i=1 to maxInput-1 do
    $\text{R}_i$ = CoverRows($\text{row}_s$, $\text{row}_s + i$, maxInput, M)
    area = totalCandidateArea($\text{row}_s$, $\text{row}_s$+i, M)
    score = area / $\text{R}_i$.size
    if score $\geq$ maxScore then
        bestRow = $\text{row}_s + i$
        rUsed = $\text{R}_i$.size
r = r - rUsed
return (bestRow+1, r)
```

```
Algorithm 5 : CoverRows (M-Bucket-I)
Input: $\text{row}_f, \text{row}_l, maxInput, M$
Regions = $\emptyset$; r=newRegion()
for all $c_i$ in M.getColumns do
    if r.cap < $\text{c}_i$.candidateInputCosts then
        Regions = Regions$\cup$r
        r=newRegion()
    r.Cells = r.Cells $\cup$ $\text{c}_\text{i}$.candidateCells
return Regions
```

We use M-Bucket-I in a binary search to find the smallest value *maxInput* for which M-Bucket-I can find a cover that uses at most $r$ regions—one per reducer. The upper and lower bound of *maxInput* for the binary search are $|S| + |T|$ and $2\sqrt{\text{number-of-candidate-cells}/r}$. The former is obvious, because we can cover the entire matrix with a rectangle of $|S|$ rows by $|T|$ columns. The latter follows from Lemma 1 and the fact that max-reducer-output is at least number-of-candidate-cells$/r$.

Recall that M-Bucket-I was designed to minimize max-reducer-input. For output-size dominated joins, one should minimize max-reducer-output instead. For this problem, we developed a heuristic called M-Bucket-O. It proceeds like M-Bucket-I, but instead of working with an input-size limit *maxInput*, it limits regions by area, i.e., number of candidate cells contained in a region.

Notice that M-Bucket-I can take better advantage of input histograms than M-Bucket-O, because it knows exactly how many input tuples from each data set belong to each bucket. On the other hand, the actual output size of a bucket could be anything between zero and the product of the bucket counts. Hence M-Bucket-I can reliably balance input-related costs even with fairly coarse-grained histograms, while M-Bucket-O can show significant output-cost imbalance even for very fine-grained histograms (e.g., where each bucket contains an average of five distinct attribute values). Our experiments support this observation.

The M-Bucket-I algorithm described in this section can be used for any theta join. For various types of joins, we can further improve it by exploiting properties of the locations of candidate cells in the join matrix. In particular, for equi-joins, band-joins and inequality joins, the join matrix has the following monotonicity property: If cell $(i, j)$ is not a candidate cell (i.e., it is guaranteed to be `false`), then either all cells $(k, l)$ with $k \leq i \wedge l \geq j$, or all cells $(k, l)$ with $k \geq i \wedge l \leq j$ are also `false` and hence not candidate cells. (Which case holds for cell $(i, j)$ is easy to determine based on the join predicate.) We can therefore find all candidate cells faster by pruning the search space based on this monotonicity property.

### 5.3 The Overall Algorithm

Given two input sets $S$ and $T$, and a join condition $\theta$, we can often choose between different MapReduce implementations. For equi-joins, there are the standard algorithm (Section 2.2), the M-Bucket algorithms, and 1-Bucket-Theta. For any other theta-join, there are the M-Bucket algorithms (if statistics are available) and 1-Bucket-Theta. For band-joins and inequality-joins, in addition to 1-Bucket-Theta, we can compute statistics as described in Section 5.1 and use M-Bucket algorithms.

Depending on the join condition, we consider all applicable algorithms. From their corresponding matrix-to-reducer mapping, we can estimate max-reducer-input and max-reducer-output for each algorithm. Then we can apply traditional cost estimation techniques from databases, because the job completion time is determined by the reducer (i.e., single processor) that receives the greatest input and the reducer that generates the greatest output. Local reducer computation is directly amenable to traditional cost analysis involving CPU and I/O cost. For DFS data transfer we can approximate cost through a disk-like model of average latency and transfer time. Details are left to future work.

### 5.4 Extension: Memory-Awareness

Given $r$ reducers, our algorithms presented in earlier sections create $r$ partitions of the join matrix to assign one partition to each reducer. Sometimes these partitions are too big to fit in memory, forcing the join algorithm to perform local disk I/O (or failure if join implementation assumes that data fits in memory). We can avoid this situation by making the algorithms "memory-aware". Instead of letting the reducer number drive the matrix partitioning, regions can be constrained to not exceed a specified input size. Given a memory limit $m$, 1-Bucket-Theta covers the entire matrix with squares of side-length $m/2$. M-Bucket-I does not perform a binary search on input size, but runs the heuristic immediately for input limit $m$, choosing more than $r$ regions if necessary. M-Bucket-O can be extended similarly.

## 6. EXPERIMENTS

We discuss representative results for our algorithms for joining real and synthetic data. All experiments were performed on a 10-machine cluster running Hadoop 0.20.2 [1]. One machine served as the head node, while the other 9 were the worker nodes. Each machine has a single quad-core Xeon 2.4GHz processor, 8MB cache, 8GB RAM, and two 250 GB 7.2K RPM hard disks. All machines are directly connected to the same Gigabit network switch. In total, the cluster therefore has 36 cores with 2GB memory per core available for Map and Reduce tasks. Each core is configured to run one map and one reduce task concurrently.

**Table 1: Skew Resistance of 1-Bucket-Theta**

| | | 1-Bucket-Theta | | Standard | |
|---|---|---|---|---|---|
| Data set | Output Size (billion) | Output Imbalance | Runtime (secs) | Output Imbalance | Runtime (secs) |
| Synth-0 | 25.00 | 1.0030 | 657 | 1.0124 | 701 |
| Synth-0.4 | 24.99 | 1.0023 | 650 | 1.2541 | 722 |
| Synth-0.6 | 24.98 | 1.0033 | 676 | 1.7780 | 923 |
| Synth-0.8 | 24.95 | 1.0068 | 678 | 3.0103 | 1482 |
| Synth-1 | 24.91 | 1.0089 | 667 | 5.3124 | 2489 |

The distributed file system block size is set to 64MB and all machines participate as storage nodes for the DFS.

We present results for the following data sets:

`Cloud`: This is a real data set containing extended cloud reports from ships and land stations [11]. There are 382 million records, each with 28 attributes, resulting in a total data size of 28.8GB.

`Cloud-5-1, Cloud-5-2`: These are two independent random samples of 5 million records each from Cloud. They are used for experiments with output-size dominated joins.

`Synth-`$\alpha$: For a fixed $\alpha$, this is a pair of data sets (one for each join input). Both contain 5 million records, each record being a single integer number between 1 and 1000. For one data set, numbers are drawn uniformly at random from this range. For the other data set, we use the Zipf distribution for the same range. Skew is adjusted by choosing a value between 0 (uniform) and 1.0 (skew) for $\alpha$, which is the usual Zipf skew parameter.

### 6.1 1-Bucket-Theta vs. Standard Equi-join

Table 1 shows results for computing an equi-join on `Synth-`$\alpha$ for various values of $\alpha$. Since all experiments have significantly larger output than input, we report output imbalance. Imbalance is computed as max-reducer-output divided by average-reducer-output. (We compute the average over 36 nodes to not favor 1-Bucket-Theta when it uses fewer reducer nodes.) We compare the load imbalance of 1-Bucket-Theta against the standard equi-join implementation from the literature (see Section 2.2).

As we discussed in Sections 4.1.1 and 4.2, 1-Bucket-Theta is virtually guaranteed to achieve an excellent balancing of both input and output tuples because of its randomized approach. As predicted, the standard algorithm's approach of creating reduce jobs based on join attribute values suffers from increasing skew. The node responsible for the most frequent value is overloaded by a factor of 3 to 5 for skewed data. This could be much worse for a join attribute with smaller cardinality even for small data set size. Since total output size is approximately the same, output imbalance directly reflects max-reducer-output. And as predicted by our qualitative cost analysis, greater max-reducer-output leads to greater job completion time.

### 6.2 Input-Size Dominated Joins

We study M-Bucket-I for the following selective self-join on the large `Cloud` data set:

```
SELECT S.date, S.longitude, S.latitude, T.latitude
FROM Cloud AS S, Cloud AS T
WHERE S.date = T.date AND S.longitude = T.longitude
      AND ABS(S.latitude - T.latitude) <= 10
```

This join produces 390 million output tuples, a much smaller set than the total of almost 800 million input records.

The statistics for M-Bucket-I are approximate equi-depth one-dimensional histograms of different granularities (i.e., number of buckets) for `Cloud`. The 1-dimensional sort key for the histogram is the combined (date, longitude, latitude)-vector, using alphabetical sorting with date as the most significant component. Notice that for a 1-bucket histogram, M-Bucket-I degenerates to a version of 1-Bucket-Theta. Instead of using the square-only approach of 1-Bucket-Theta (which was needed to achieve the bounds in Theorem 3, but often does not use all reducers), we run the heuristic described in Section 5.2 to find the matrix-to-reducer mapping.

For coarse histograms (1 to 100 buckets), max-reducer-input size turned out to exceed main memory size, while this was not the case for more fine-grained histograms. To make results more comparable, we therefore used our **memory-aware version** of M-Bucket-I for all experiments. Note that this version will create $c \cdot r$ regions for $r$ reducers and some appropriately chosen integer $c \geq 1$, just large enough to make the computation fit into reducer memory. More regions mean greater input duplication. We report input duplication rate as the ratio of the total number of Map output records and the total number of Map input records. Input duplication rate for histograms with 1, 10, 100, 1000, 10000, 100000, and 1000000 buckets are 31.22, 8.92, 1.93, 1.0426, 1.0044, 1.00048, and 1.00025 respectively.

Figure 6 shows how input imbalance, computed as (max-reducer-input / avg-reducer-input), of M-Bucket-I changes with varying histogram granularity. (Notice that all 36 reducers were used in all cases, hence avg-reducer-input is comparable across algorithms.) The graph shows that our heuristic achieves its goal of balancing input evenly across reducers. For 1 bucket, this is achieved automatically by the randomization of 1-Bucket-Theta. For histograms with more than one bucket, randomization is limited to happen within buckets and imbalance can be caused if the input size for a matrix region assigned to a reducer is not estimated accurately.

Figure 7 compares max-reducer-input for M-Bucket-I as we vary the granularity of the input statistics. Here the data point for 1 bucket again corresponds to the version of 1-Bucket-Theta described above. It is obvious that even though 1-Bucket-Theta almost perfectly balances input load, it has a lot more to balance than M-Bucket-I with more fine-grained histograms. The reason is that 1-Bucket-Theta covers the entire join matrix, while M-Bucket-I can avoid covering a large fraction of the matrix based on the available statistics and properties of the band-join. As predicted by our cost analysis, Figure 8 shows for this input-size dominated join that MapReduce job completion time tracks the trend of max-reducer-input almost perfectly.

Figure 8 does *not* include M-Bucket-I's pre-processing costs. The detailed cost breakdown and true total job completion times are shown in Table 2. Average run times of 10 executions are reported for each experiment. The maximum standard deviation between the job completion times was 14.61%. In addition to performing the MapReduce job for the join, M-Bucket-I first has to compute the statistics (unless they are available) by finding the approximate quantiles (single pass to compute random sample) and then counting the number of records per quantile range (another sequential pass). The heuristic for finding the best matrix-to-reducer mapping is run on a single node, followed by the actual join.

**Table 2: M-Bucket-I cost details (seconds)**

| Step | Number of Buckets | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
| Quantiles | 0 | 115 | 120 | 117 | 122 | 124 | 122 |
| Histogram | 0 | 140 | 145 | 147 | 157 | 167 | 604 |
| Heuristic | 74.01 | 9.21 | 0.84 | 1.50 | 16.67 | 118.03 | 111.27 |
| Join | 49384 | 10905 | 1157 | 595 | 548 | 540 | 536 |
| Total | 49458.01 | 11169.21 | 1422.84 | 860.5 | 843.67 | 949.03 | 1373.27 |

As we can see, at some point more fine-grained statistics improve join cost only minimally, while especially the heuristic for finding the matrix-to-reducer mapping becomes more expensive as more (and smaller) regions of candidate cells have to be considered by M-Bucket-I.

In summary, this experiment highlights the Achilles heel of 1-Bucket-Theta—its potentially high input duplication. Input duplication was worse than the number of reducers (set to 36) suggests. The memory-aware version created up to 972 partitions to guarantee that all reducer computations can be performed in memory. For smaller input data or when running the version with local I/O (and hence fewer partitions), input duplication would be significantly lower. The experiments also show that whenever available statistics and properties of the join condition enable us to avoid mapping a large fraction of the join matrix to any reducer, input-related costs can be reduced significantly.

## 6.3 Output-Size Dominated Joins

We study the following modestly selective join on the smaller `Cloud-5` real data sets:

```
SELECT S.latitude, T.latitude
FROM Cloud-5-1 AS S, Cloud-5-2 AS T
WHERE ABS(S.latitude - T.latitude) <= 2
```

This join produces 22 billion output tuples, a much larger set than the total of 10 million input records.

This experiment mirrors the one reported in the previous section, but now for a join that is output-size dominated. As with M-Bucket-I, M-Bucket-O for a histogram with only one bucket degenerates to a version of 1-Bucket-Theta. It tries to find a region partitioning of the join matrix where all regions have approximately the same area (number of candidate cells covered), but no reducer is left without work (as could happen for the original 1-Bucket-Theta algorithm, whose primary goal was to guarantee the result in Theorem 3).

Figure 9 shows the output-size imbalance, computed as (max-reducer-output / avg-reducer-output), for different granularities of the histogram. (Notice that all 36 reducers were used in all cases, hence avg-reducer-output is comparable across algorithms.) 1-Bucket-Theta, as expected, achieves almost perfect output balancing due to its randomized approach. However, for other histogram granularities, output imbalance is much greater than the corresponding numbers for the *input* imbalance for M-Bucket-I for the previous experiment. Even though input sizes can be estimated well, even from comparably coarse-grained histograms (see Figure 6), the number of output tuples in a region is difficult to estimate because join selectivity can vary significantly in different regions. 1-Bucket-Theta is practically immune to this problem, because its randomization "shuffles" tuples around randomly, hence tuples from regions with high result density and those from regions with low result density

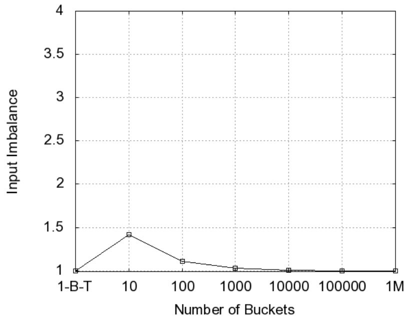


**Figure 6: Input imbalance for 1-Bucket-Theta (#buckets=1) and M-Bucket-I on `Cloud`**

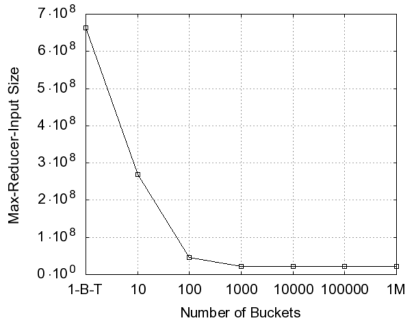


**Figure 7: Max-reducer-input for 1-Bucket-Theta and M-Bucket-I on `Cloud`**

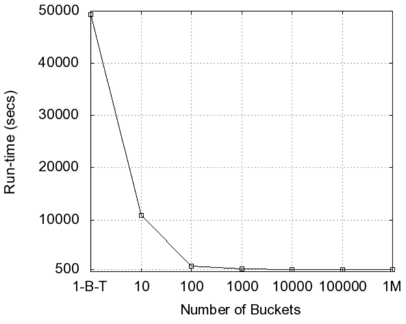


**Figure 8: MapReduce time for 1-Bucket-Theta and M-Bucket-I on `Cloud`**

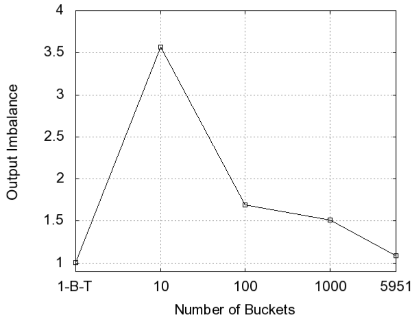


**Figure 9: Output imbalance for 1-Bucket-Theta (#buckets=1) and M-Bucket-O on `Cloud-5`**

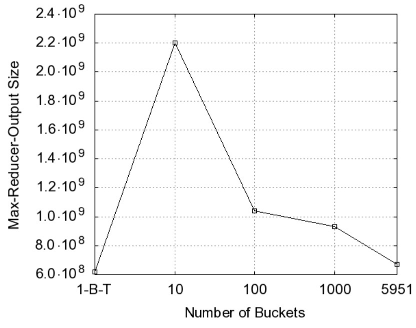


**Figure 10: Max-reducer-output for 1-Bucket-Theta and M-Bucket-O on `Cloud-5`**

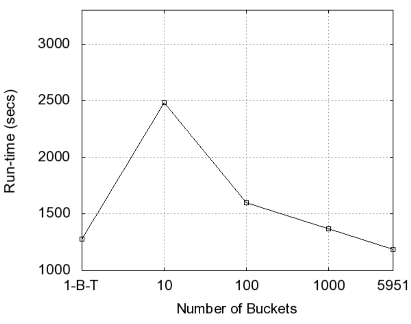


**Figure 11: MapReduce time for 1-Bucket-Theta and M-Bucket-O on `Cloud-5`**

**Table 3: M-Bucket-O cost details (seconds)**

| Step | Number of Buckets | | | | |
|---|---|---|---|---|---|
| | 1 | 10 | 100 | 1000 | 5951 |
| Quantiles | 0 | 4.52 | 4.54 | 4.8 | 4.9 |
| Histogram | 0 | 26.2 | 25.8 | 25.6 | 25.6 |
| Heuristic | 0.04 | 0.04 | 0.05 | 0.24 | 0.81 |
| Join | 1278.6 | 2483.4 | 1596.6 | 1368.8 | 1188 |
| Total | 1278.64 | 2514.16 | 1626.99 | 1399.44 | 1219.31 |

in the join matrix get intermixed. M-Bucket-O cannot do this, because a tuple can only be assigned to the appropriate histogram bucket, not any random bucket.

Since the total join output size is independent of the number of histogram buckets, Figure 10 looks exactly the same as Figure 9, just with all numbers scaled by average-reducer-output. And since the join is output-size dominated, the job completion time numbers in Figure 11 for the MapReduce join implementation closely track the values for max-reducer-output. This highlights that minimizing max-reducer-output for this problem is the right approach. The true total job completion times are listed in detail in Table 3. Average run times of 10 executions are reported for each experiment. The maximum standard deviation between the job completion times was 3.42%. Input duplication rate for histograms with 1, 10, 100, 1000, and 5951 buckets are 7.50, 4.14, 1.46, 1.053, and 1.0349 respectively.

In summary, for the output-size dominated join problem, 1-Bucket-Theta performed better than M-Bucket-O, except when very detailed statistics were available for the latter. Notice that there are 5951 different latitude values in the data set, hence the histogram with 5951 had a single bucket per occurring latitude value. This allowed M-Bucket-O to compute exact output sizes for any region in the join matrix. With fewer buckets, estimation error increased significantly, resulting in worse output balancing.

## 7. RELATED WORK

Most previous work implements equi-joins in MapReduce as described in Section 2.2 for data sets that do not fit in memory [4, 5, 14, 16, 20]. Map-Reduce-Merge [20] supports other joins and different implementations, but it requires an extension of the MapReduce model. Users also have to implement non-trivial functions that manipulate the dataflow in the distributed system. Our approach does not require any change to the MapReduce model, but still supports any theta-join in a single MapReduce job. Hence it is possible to integrate our approach with high-level programming languages on top of MapReduce [2, 15, 17].

[4] studies multiway equi-joins in MapReduce, optimizing for throughput by selecting a query plan with the lowest input replication cost. It is not clear if these results would generalize to other join conditions. Vernica et al. [19] present an in-depth study of a special type of similarity join in MapReduce. Some clever techniques for dealing with memory limits are proposed. To the best of our knowledge, our paper

is the first to study all theta-joins and explore optimality properties for them in MapReduce-based systems.

An overview of parallel join algorithms studied can be found in [10]. For parallel non-equi joins, [18] fragments both data sets and replicates them among processors so that each item from one input meets each item from the other. The replication rate of inputs are decided using heuristics to minimize total communication cost. Partitioning found by these methods can be integrated into our approach and used in MapReduce. Earlier work by DeWitt et al. [8] explored how to minimize disk accesses for band-joins by choosing partitioning elements using sampling. The parallel version of this algorithm could be modeled by a join matrix, allowing us to include the approach into our framework when choosing the algorithm with the best input and/or output balancing properties.

Data skew was shown to cause both poor query cost estimates and sub-linear speedup [7]. Large scale data analysis platforms try to address skew in computation times through speculative execution for tasks which are expected to dominate end-to-end latency [6, 12]. This approach does not handle data skew in joins, because the excessively large tasks would just be executed on another machine, but not broken up. Kwon et al. [13] attempt to minimize computational skew in scientific analysis tasks with blackbox functions. In order to deal with skew, DeWitt et al. [9] propose four equi-join algorithms and show that traditional hybrid hash join is the winner in lower skew or no skew cases. Pig [3] supports these skewed equi-join implementations on top of MapReduce. We demonstrated that we can handle skew not only for equi-joins, but any arbitrary join conditions.

## 8. CONCLUSIONS

We proposed algorithms for implementing any theta-join as a single MapReduce job. This implementation is achieved by creating the appropriate Map and Reduce functions, without any modifications to the MapReduce framework.

Starting with the goal of minimizing total job completion time, we showed how to define a great variety of join implementations using appropriate join matrix-to-reducer mappings. We proposed 1-Bucket-Theta, an algorithm whose matrix-to-reducer mapping we showed to be provably close to optimal for the cross-product and for any join whose output is a significant fraction of the cross-product. For more selective join conditions we showed that even though there might be faster algorithms than 1-Bucket-Theta, in practice it might often not be possible to identify these algorithms as usable without performing expensive analysis or without knowing the join result in advance.

We proposed the M-Bucket class of algorithms that can improve runtime of any theta-join compared to 1-Bucket-Theta by exploiting input statistics to exclude large regions of the join matrix with a comparably lightweight test, and thus reduce input-related costs.

Interesting future directions include the evaluation of multi-way theta-joins and the development of a complete optimizer for selecting the best MapReduce implementation for any given join problem.

## 9. REFERENCES

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] Apache hive. `http://hadoop.apache.org/hive`.

[3] Apache pig. `http://pig.apache.org/`.

[4] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[5] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986, 2010.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[7] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[8] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, pages 443–452, 1991.

[9] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–40, 1992.

[10] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25, 1993.

[11] C. Hahn and S. Warren. Extended edited synoptic cloud reports from ships and land stations over the globe, 1952-1996. `http://cdiac.ornl.gov/ftp/ndp026c/ndp026c.pdf`.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[13] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SoCC*, 2010.

[14] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX*, pages 267–273, 2008.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[16] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[17] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[18] J. W. Stamos and H. C. Young. A symmetric fragment and replicate algorithm for distributed joins. *IEEE Trans. Parallel Distrib. Syst.*, 4:1345–1354, 1993.

[19] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

[20] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: Simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, 2007.