A Guide to Formal Analysis of Join Processing in Massively Parallel Systems

Paraschos Koutris
University of Wisconsin-Madison
paris@cs.wisc.edu

Dan Suciu
University of Washington
suciu@cs.washington.edu

1. INTRODUCTION

Over the last decade, there has been an enormous increase in the volume of data that is being stored, processed and analyzed. In order to improve the performance of query processing on such amounts of data, many modern data management systems (*e.g.* Spark [23, 28], Hadoop [13, 9, 24], and others [19, 14]) have resorted to the power of parallelism to speed up computation. Parallelism enables the distribution of computation for data-intensive tasks into hundreds, or even thousands of machines, and thus significantly reduces the completion time for several crucial data processing tasks.

In this paper, we present a survey on recent results [18, 4, 5, 17] that study the computational complexity of mulitway join processing in such massively parallel systems. Our goal is twofold. First, we introduce a simple theoretical model, called the MPC (Massively Parallel Computation) model, that allows us to rigorously analyze the computational complexity of various parallel algorithms for query processing. Second, using the MPC model as a theoretical tool, we show how we can design novel algorithms and techniques for multiway join processing, and how we can prove their optimality through tight lower bounds. Our analysis provides a deeper understanding of how much synchronization, communication and data load is required when we compute a multiway join query, and informs of what is possible to achieve under specific system constraints.

Organization. We first present the MPC model in Section 2. Equipped with the model, we describe and rigorously analyze the behavior of several algorithms for the natural join query (Section 3) and the triangle query (Section 4). These two queries cover most of the techniques and algorithms we use for general multiway join queries, which we subsequently describe in Sections 5 and 6. We conclude in Section 7 by discussing some key takeaways of our results.

2. THE MPC PARALLEL MODEL

We introduce here the *Massively Parallel Computation* model, or MPC. We will use MPC to analyze the parallel complexity of various multiway join algorithms, as well as prove lower bounds on the amount of communication and synchronization.

In the MPC model, computation is performed by a cluster of p machines using a shared-nothing architecture. The *shared-nothing* paradigm is widely applied in modern big data management systems [25]. The computation proceeds in *rounds*: each round consists of some local computation followed by global exchange of data between the machines. At the end of each round, the machines have to *synchronize*, *i.e.* wait for all machines to finish before proceeding to the next round.

The input data of size m (in tuples) is initially evenly distributed among the p machines, *i.e.* each machine stores m/p data (this captures how input relations are typically distributed in a distributed file system like HDFS [22]). After the computation is complete, the output result is present in the union of the output of the p machines.

The complexity of an algorithm in the MPC model is characterized by two parameters:

number of rounds (r) This parameter captures the number of synchronization points that an algorithm requires during execution. A smaller number of rounds means that the algorithm can run with less synchronization.

maximum load (L) This parameter is defined as the maximum amount of data that a machine can *receive* at any round during computation. A smaller load means that data is more evenly distributed, and the amount of data communicated is smaller.

An ideal algorithm uses a single round ($r = 1$) and distributes the data evenly without any replication, hence achieving maximum load $L = m/p$. Since this is rarely possible, algorithms for query

processing need to use more rounds, have an increased maximum load, or both. An algorithm with load $L = m$ does not exploit data parallelism at all, since we can send all input data to a single machine and do processing locally. In general, for the problems we will discuss in this paper, the load will be of the form $L = m/p^{1-\varepsilon}$, for some $0 \leq \varepsilon < 1$; we call the parameter ε the *space exponent*. The challenge is to identify the optimal tradeoff between the number of rounds and maximum load for various computational tasks.

Other Parallel Models. There has been a plethora of parallel computation models proposed over the years [1, 15, 11]. The MPC model is closer to the BSP model [26], which also describes synchronous computation; the main difference is that MPC ignores the computation cost as a parameter, and focuses instead on the amount of data communicated at each machine.

3. JOINS IN PARALLEL

We first present how we can compute a natural join between two binary relations,

$$J(x, y, z) = R(x, y), S(y, z).$$

Let m_R, m_S be the sizes of R, S respectively, and assume that initially both R and S are uniformly distributed on the p machines in the cluster; thus, the input load per machine is m/p, with $m = m_R + m_S$. We discuss several algorithms that all work using a single round, and analyze in detail the load L they can achieve.

3.1 Hash Join

Let h be a random hash function that maps attribute values (U) to the domain $[p] = \{1, \cdots, p\}$. To partition the tuples, every machine iterates over its local tuples and sends every tuple $R(a, b)$ to machine $h(b)$, and every tuple $S(c, d)$ to machine $h(c)$. After receiving the tuples, each machine computes the join locally. This basic one-round algorithm was pioneered since the earliest parallel database systems [10] and can be found today in virtually all parallel join implementations [9, 19, 7, 29, 27].

Load Analysis. Since we are distributing m tuples over p machines, one may hope the load to be m/p; unfortunately, this is not always the case. We rule out bad hash functions with many collisions: there is a lot of work on designing good hash functions and we assume to have one. In particular, we will assume that h is a *perfectly random hash function*: this means that h is drawn uniformly at random from the set of hash functions that map the universe U to $[p]$. Such a hash function guarantees that (i) for any value $v \in U$ and every hash bucket $s \in [p]$, $P(h(x) = s) = 1/p$, and (ii) for any distinct values $v_1, v_2, \ldots$, the hash buckets $h(v_1), h(v_2), \ldots$ are independent [20, pp.107].[1]

Observation 1. *The expected load for any machine s is m/p, since each input tuple in R or S is sent to the machine with probability $1/p$ over the random choices of the hash function.*

Define the *degree* d_i of a value $v_i \in U$ as the number of tuples $R(x, y)$ or $S(y, z)$ with $y = v$. We say that the value is *skewed* if its degree is $> m/p$.

Observation 2. *If the input has a skewed value, then for any choice of hash function h, there exists an* overloaded *machine with load $> m/p$. Indeed, all input tuples having the skewed value must be hashed to the same machine, which becomes overloaded.*

Thus, the ideal load is m/p, but if the input is skewed then the load exceeds m/p. It turns out that the converse also holds: if the database is skew-free, then the maximum load is $O(m/p)$ with high probability. However, in the rigorous analysis we must pay an additional $\ln(p)$ factor, either by allowing the load to grow to $\ln(p) \cdot m/p$, or by requiring the maximum degree to be $< m/(p \cdot \ln(p))$. The rest of this subsection provides the full formal analysis, for readers interested in the detailed argument.

Suppose $U = \{v_1, \ldots, v_N\}$ is the universe of possible values; let $d_i = 0$ if v_i does not occur in the input. The input size is $m = \sum_i d_i$.

Theorem 3.1. *[6, 12] Let $X_1, \ldots, X_N \in \{0, 1\}$ be independent and identically distributed (i.i.d.) random variables such that $P(X_i = 1) = 1/p$. Let $d = \max_{i=1}^N d_i$. Then, for any $\delta \geq 0$:*

$$P\left(\sum_{i=1}^{N} d_i X_i \geq (1+\delta)m/p\right) \leq \exp\left(-\frac{m}{pd}h(\delta)\right)$$

where $h(\delta) = (1+\delta)\ln(1+\delta) - \delta$.

We use Theorem 3.1 to analyze the load of the Hash Join algorithm. Fix some machine, and denote X_i the random variable that is 1 if the value v_i is hashed to that server, and 0 otherwise. Then $P(X_i = 1) = 1/p$, and the load at that machine is $\sum_i d_i X_i$. The probability that the maximum load L, over all machines, exceeds the expected load m/p

[1] In practice, we can choose a good enough hash function, for example by having a fixed function h_0, choosing a random seed r, and define $h(v) = h_0(v \text{ xor } r)$.

by a factor $1+\delta$ follows from Theorem 3.1 and the union bound:

$$P(L > (1+\delta)m/p) \leq p \exp\left(-\frac{m}{pd}h(\delta)\right)$$

The main result for Hash Join follows:

PROPOSITION 3.2. *The maximum load L for the Hash Join algorithm is bounded as follows:*

(1) If $d = 1$ (the degree of every value is 1), then for any $\delta < 1$, $P(L > (1+\delta)m/p) \leq p\exp(-\frac{m\delta^2}{3p})$; this probability decreases exponentially fast in the input size m.

(2) If $d \leq m/(3p\ln(p))$, then for any $\delta > 1$, $P(L > (1+\delta)m/p) \leq p^{1-\delta}$; this probability decreases polynomially in the number of machines p.

(3) If $d \leq m/p$, $P(L > \ln(p)m/p) \leq p^{-1}$; this probability also decreases polynomially in the number of machines p.

PROOF. Item (1) follows from the fact that for $\delta < 1$, we have that $h(\delta) \geq \delta^2/3$. Item (2) follows from $h(\delta) \geq \delta/3$ and $p\exp(-\ln(p)\delta) = p^{1-\delta}$. Item (3) follows from setting $1+\delta = \ln p$. Then, since $(1+\delta)\ln(1+\delta) - \delta > 2(1+\delta)$ whenever $\ln(1+\delta) > 3$, we have $p\exp(-2\ln(p)) = p^{-1}$. □

Even though the above analysis is for the case of a simple join, all algorithms in this paper that use one or more hash functions to partition data values into buckets can also be analyzed in a similar way, using a generalization of Theorem 3.1. The main takeaway is that skew-free input means that the maximum load is close to the expected load; on the other hand, skewed input can cause the maximum load to be much larger than the expected load. The particular threshold that determines when a value becomes skewed depends on (i) the query, and (ii) the sizes of the relations.

3.2 Broadcast Join

A simple join algorithm that is immune to skew is the Broadcast Join, where each machine *broadcasts* all its local S-tuples to all other machines. There is no need to reshuffle R, and after the communication step all machines can compute the join locally. This algorithm is also implemented in several systems [21], and performs best when the size of S is much smaller than that of R. The load is $m_R/p + m_S$, regardless of whether the data is skewed or not; this becomes $O(m/p)$ when $m_S = O(m_R/p)$.

3.3 Cartesian Join

The worst behavior for the Hash Join algorithm occurs when both R and S exhibit worst-case skew, *i.e.* they have a single y-value. Then the algorithm will send all tuples of R and S to the same machine, whose load will be $L = m$. In this case, the join degenerates to a *cartesian product*: $R(x) \times S(z)$ (we drop the attribute y since it is constant), and requires a different algorithm.

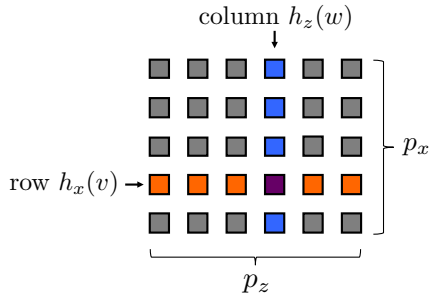

Figure 1: Depiction of the Cartesian Join algorithm. The p machines are organized into a $p_x \times p_z$ rectangle.

Let p_x, p_z be two integers such that $p_x \cdot p_z = p$; we call these numbers *shares* [2]. We organize the p machines into a $p_x \times p_z$ rectangle, where each server is uniquely identified by a pair of numbers $(i,j) \in [p_x] \times [p_z]$ (see Figure 1). Let h_x, h_z be two random hash functions with domains $[p_x]$ and $[p_z]$ respectively. The Cartesian Join works as follows. Initially, it sends every tuple $R(v)$ to all machines of the form $(h_x(v), *)$, and every tuple $S(w)$ to all machines $(*, h_z(w))$. In other words the algorithm broadcasts $R(v)$ to the entire row $h_x(v)$, and broadcasts $S(w)$ to the entire column $h_z(w)$, as seen in Figure 1. After the communication step, each machine computes the cartesian product of its local tuples. The algorithm is correct, because every answer (v,w) of the cartesian product will be in the output of some machine, namely the machine at $(h_x(v), h_z(w))$. There is no skew, since every data value has degree 1 (R and S are sets).

Load Analysis. Since R is partitioned into p_x buckets, each machine receives $O(m_R/p_x)$ tuples from R, and similarly $O(m_S/p_z)$ tuples from S. The load is then

$$L = \frac{m_R}{p_x} + \frac{m_S}{p_z} \geq 2\left(\frac{m_R m_S}{p_x p_z}\right)^{1/2} = 2\left(\frac{m_R m_S}{p}\right)^{1/2}$$

where equality holds when the two terms m_R/p_x, m_S/p_z are equal. This implies that the optimal values for p_x, p_z are:

$$p_x = \left(p\frac{m_R}{m_S}\right)^{1/2} \qquad p_z = \left(p\frac{m_S}{m_R}\right)^{1/2}$$

When $m_R = m_S$, the algorithm organizes the p machines in a $\sqrt{p} \times \sqrt{p}$ square. Otherwise, it allocates

more shares to the larger relation. Since we have to ensure that $p_x, p_z \geq 1$, if $m_R < m_S/p$, we set $p_x = 1, p_z = p$, in which case the algorithm degenerates to the Broadcast Join (R is broadcast) and the load becomes $O(m_S/p)$. Similarly, if $m_S < m_R/p$, we set $p_x = p, p_z = 1$ and the load becomes $O(m_R/p)$. The load of the Cartesian Join will be:

$$L = O\left(\max\left\{\frac{m_R}{p}, \frac{m_S}{p}, \left(\frac{m_R m_S}{p}\right)^{1/2}\right\}\right) \quad (1)$$

We should note that so far we have ignored any rounding issues for the (integer) shares; rounding is challenging problem in practice, and we refer the reader to [8] for further discussion.

Lower Bound. It is easy to see that the above allocation of shares is optimal. We will provide a brief sketch of the argument. Suppose that machine j receives $m_{R,j}, m_{S,j}$ tuples from R and S respectively. Then, it can output at most $m_{R,j} \cdot m_{S,j} \leq (m_{R,j} + m_{S,j})^2/4 = L_j^2/4$ tuples. Since we have p machines, the total output $\sum_j L_j^2/4$ must be at least $m_R m_S$. Therefore,

$$m_R m_S \leq \sum_j L_j^2/4 \leq \sum_j L^2/4 = pL^2/4$$

where the last inequality follows from $L = \max_j L_j$. Thus, $L \geq 2(m_R m_S)^{1/2}/p^{1/2}$.

3.4 Skew Join

All algorithms discussed so far achieve the optimal load only if the data has no skew. To achieve the optimal load in the case of skew, we follow a different approach: we treat the heavy (skewed) and light (non-skewed) values separately.

Heavy Hitters. Recall that the analysis of the Hash Join algorithm fails if some value $y = v$ occurs more than m/p times in R and S, in which case we say it is a *heavy hitter*. We explain briefly how to compute these values. Fix a machine, and call a local value v a *candidate* if its local degree is $> m/p^2$. Each machine computes its candidates and their local degrees. All candidates are broadcast to all machines, and then each machine computes the global degrees of all candidates by adding up the local degrees; the heavy hitters are those candidates with a global degree $> m/p$. Although we need one round to compute the heavy hitters, we will not count this step towards the total number of communication rounds, because the size of data exchange is much smaller, and in many cases the heavy hitters are known already.

The Skew Join Algorithm. Let $\mathcal{H}$ be the set of heavy hitters, and d_h^R, d_h^S be the degrees of $y = h$ in $R(x, y)$ and $S(y, z)$ respectively. Notice that value h may be skewed only in R, or only in S, or in both. Skew Join computes in parallel (1) the join on the light hitters, and (2) the join on the heavy hitters.

The light hitters are computed using the Hash Join; Proposition 3.2 tells us that the load will be $O(\ln(p)m/p) = \tilde{O}(m/p)$; we will use $\tilde{O}$ to hide any logarithmic factors. For the heavy hitters, we assign to each value $h \in \mathcal{H}$ an exclusive group of p_h machines such that $p = \sum_h p_h$, which we use to compute the residual query corresponding to the heavy hitter value $y = h$. The *residual query* $q[h/x]$ is obtained by substituting x with the constant h; in this case, it is the cartesian product $R(x, h) \times S(h, z)$. According to Eq. (1), we can compute $q[h/x]$ with load $L_h = \max\{d_h^R/p_h, d_h^S/p_h, \left(d_h^R d_h^S/p_h\right)^{1/2}\}$. Since the maximum load is $L = \max_h L_h$, to minimize L we choose the p_h to make all L_h equal:

$$p_h = p \cdot \max\left\{\frac{d_h^R}{m_R}, \frac{d_h^S}{m_S}, \frac{d_h^R d_h^S}{\sum_j d_j^R d_j^S}\right\}$$

The maximum load of Skew Join becomes:

$$L = \tilde{O}\left(\max\left\{\frac{m_R}{p}, \frac{m_S}{p}, \sqrt{\frac{\sum_h d_h^R d_h^S}{p}}\right\}\right)$$

By extending the argument from the previous section, it can be shown that this load is optimal, given the degree distribution for d_h^R, d_h^S.

Discussion. We end this section by discussing two special cases of interest. The first is when there is a single heavy hitter h which occurs in all tuples in R and in S, in other words $d_h^R = m_R$, $d_h^S = m_S$. In this case $p_h = p$, and Skew Join degenerates to a Cartesian Join. In this scenario of extreme skew, the speedup is reduced from linear ($\tilde{O}(m/p)$) to sublinear ($\tilde{O}(m/p^{1/2})$). The second special case is when the skew is one-sided, *i.e.* when only values in R are skewed. Then $d_h^S \leq m_S/p$ for all h, and the reader can verify that the load becomes $\tilde{O}(\max\{m_R/p, m_S/p\}) = \tilde{O}(m/p)$. In other words, a join with one-sided skew is no more expensive than a skew free-join, and has linear speedup.

4. TRIANGLES IN PARALLEL

Our next query is the triangle query:

$$\Delta(x, y, z) = R(x, y), S(y, z), T(z, x).$$

We denote the sizes of R, S, T by m_R, m_S, m_T respectively. A traditional parallel query execution engine typically computes the triangle query in two rounds. In the first round, it computes the natural join $U = R(x, y), S(y, z)$ using the parallel Hash

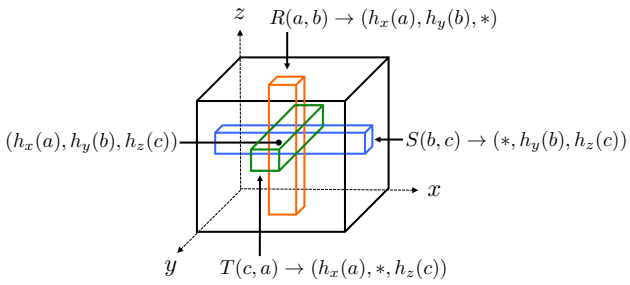

Figure 2: Depiction of the HC algorithm for the triangle query. The p machines are organized into a $p_x \times p_y \times p_z$ cube.

Join algorithm. In the second round, it joins the intermediate relation $U(x, y, z)$ with $T(z, x)$ using again a parallel Hash Join. The issue with such an execution strategy is that the intermediate relation U can be much larger than the input, and shuffling U to join with T can be a costly operation in terms of communication. Motivated by this, we present here two one-round algorithms that compute Δ: one for skew-free input, and the other for skewed data.

4.1 The HyperCube Algorithm

The HyperCube (HC) algorithm was first introduced by Afrati and Ullman [2] in the MapReduce context, with the name SHARES. It first organizes the p machines into a 3-dimensional cube (one dimension per variable). Let p_x, p_y, p_z be the sizes of the dimension for variables x, y, z respectively, called *shares*. Each machine is represented by a distinct point in $\mathcal{P} = [p_x] \times [p_y] \times [p_z]$. Since we have p available machines, we have $p_x \cdot p_y \cdot p_z \leq p$.

The algorithm makes use of three hash functions h_x, h_y, h_z, which map values from the domain U to $[p_x]$, $[p_y]$, $[p_z]$ respectively. During the communication phase of the first round, each tuple $R(a, b)$ is sent to all machines of the form $(h_x(a), h_y(b), *)$. Notice that every tuple is replicated p_z times, since it is sent to p_z machines. We distribute the tuples from S, T similarly: $S(b, c)$ is sent to $(*, h_y(b), h_z(c))$ and $T(c, a)$ to $(h_x(a), *, h_z(c))$. The communication pattern is depicted in Figure 2. During the computation phase, each machine simply performs a local computation of the query Δ on the data fragments it has received.

The correctness of the algorithm comes from the fact that any output triangle $\Delta(a, b, c)$ will be computed and output on the machine with coordinates $(h_x(a), h_y(b), h_z(c))$, since that machine will have all necessary input data sent to it.

Load Analysis. The above algorithm works in one round, and outputs the correct result. We next analyze how to choose the shares and how to compute the maximum load L.

We first focus on how R is distributed. The key observation is that R is partitioned into $p_x \cdot p_y$ buckets. Assuming that R is skew-free (here this means that the degree of each x-value is $\leq m_R/p_x$ and of each y-value $\leq m_R/p_y$), each bucket will be of approximately the same size, and thus the load sent to a machine will be $\tilde{O}(m_R/(p_x p_y))$. Similarly, the load for S will be $\tilde{O}(m_S/(p_y p_z))$, and for T it will be $\tilde{O}(m_T/(p_x p_z))$. The total load L will be at least the maximum of the three quantities. To find the optimal shares, we can construct an optimization problem, where the objective is to minimize L under the constraints that $L \geq m_R/(p_x \cdot p_y)$ (similarly for S, T), and also $p_x \cdot p_y \cdot p_z \leq p$.

In order to solve the above optimization program, we transform it into a linear program (LP) by taking logarithms with base p on both sides. Let $\lambda = \log_p L$ and $e_i = \log_p p_i$ for $i = \{x, y, z\}$. Since $p_i = p^{e_i}$, we call e_i the *share exponent*. The LP takes the following form:

$$\begin{aligned} \textbf{minimize} \quad & \lambda \\ \textbf{subject to} \quad & e_x + e_y + \lambda \geq \log_p m_R \\ & e_y + e_z + \lambda \geq \log_p m_S \\ & e_x + e_z + \lambda \geq \log_p m_T \\ & e_x + e_y + e_z \leq 1 \\ & e_x, e_y, e_z, \lambda \geq 0 \end{aligned}$$

The solution of the above LP obtains the best possible share exponents for the particular sizes of the input relations for the skew-free case. By using the principle of LP duality, the optimal load can also be expressed as the maximum value of the following (non-linear) optimization problem:

$$\begin{aligned} \textbf{maximize} \quad & \left(\frac{m_R^{u_R} m_S^{u_S} m_T^{u_T}}{p} \right)^{1/(u_R+u_S+u_T)} \\ \textbf{subject to} \quad & u_R + u_S \leq 1 \\ & u_S + u_T \leq 1 \\ & u_R + u_T \leq 1 \\ & u_R, u_S, u_T \geq 0 \end{aligned}$$

The vector (u_R, u_S, u_T) forms a *fractional edge packing* of the query Δ; we will explain in the next section how this notion is defined for any multiway join query. By examining the vertices of the polytope formed by the edge packings, we can show that

the optimal load is:

$$L = \tilde{O}\left(\max\left\{\frac{m_R}{p}, \frac{m_S}{p}, \frac{m_T}{p}, \frac{(m_R m_S m_T)^{1/3}}{p^{2/3}}\right\}\right)$$

The first three terms are obtained through the edge packings $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$ respectively, while the last term through $(1/2,1/2,1/2)$.

Discussion. To keep the discussion simple, let us consider the case where $m_R < m_S = m_T = m$. There are two cases:

- $p < m/m_R$. In this case, the optimal packing is either $(0,1,0)$, or $(0,0,1)$, and the load becomes $\tilde{O}(m/p)$. To achieve this linear speedup, the HC algorithm allocates shares $p_x = p_y = 1$, $p_z = p$, *i.e.* it performs a parallel Hash Join of S, T on z, and broadcasts R.
- $p \geq m/m_R$. In this case, the optimal packing is $(1/2,1/2,1/2)$, and the load becomes $L = (m_R m_S m_T)^{1/3}/p^{2/3}$. To achieve this load, the HC algorithm allocates shares as follows: $p_x = p_y = (m_R/m)^{1/3}p^{1/3}$, $p_z = (m/m_R)^{2/3}p^{1/3}$.

In the case where all input relations have the same size, *i.e.* $m_R = m_S = m_T = m$, the shares are equal: $p_x = p_y = p_z = p^{1/3}$, and the load becomes $\tilde{O}(m/p^{2/3})$. The speedup now is not linear; however, as we will see in Section 5, this is the best we can hope for among one-round algorithms.

4.2 Triangles with Skew

The HC algorithm is optimal only for skew-free data. But how do we define skewed values (heavy hitters) for the triangle? For the sake of simplicity, assume that all relations have size equal to m. A value for x, y or z is a *heavy hitter* if it has degree $> m/p^{1/3}$ for any of the two relations it belongs; otherwise, it is light. To achieve optimal load for skewed data, we follow the same approach as the Skew Join algorithm, by treating heavy and light values separately.

The algorithm will deal with the light values by running the vanilla HC algorithm, achieving maximum load $\tilde{O}(m/p^{2/3})$. To handle the heavy hitters, we distinguish two cases.

Case 1. In this case, we handle the tuples that have values with degree $\geq m/p$ in at least two variables. Without loss of generality, suppose that both x, y are heavy in at least one of the two relations they belong to. The observation is that there are at most $2p$ such heavy values for each variable, and hence we can send all tuples in R with both x, y heavy (at most $4p^2$) to all machines. Then, it remains to compute the query $S'(y,z), T'(z,x)$, where x and y can take only p values. We can do this by running the standard Hash Join algorithm; since the degree of z-values will be at most p for each relation, there is no skew and the maximum load will be $\tilde{O}(m/p)$.

Case 2. In this case, we handle the remaining output: this includes the tuples where one value has degree $\geq m/p^{1/3}$, and the other values have degree $\leq m/p$. Without loss of generality, assume that we want to compute the query for the x-values that are heavy in either R or T. Observe that there are at most $2p^{1/3}$ such heavy hitters. If $\mathcal{H}_x$ denotes the set of heavy hitter values for variable x, the residual query $q[h/x]$ for each $h \in \mathcal{H}_x$ is:

$$q[h/x] = R(h,y), S(y,z), T(z,h)$$

which is equivalent to computing the query $q_x(y,z) = R'(y), S(y,z), T'(z)$ with input sizes d_h^R, m, d_h^T respectively. As with the Skew Join, we allocate an exclusive group of p_h servers to compute $q[h/x]$ for each $h \in \mathcal{H}_x$. Having Case 1 ensures that the input to the residual query is skew-free, hence the load L_h for a particular value h is given by:

$$L_i = O\left(\max\left\{\frac{m}{p_h}, \sqrt{\frac{d_h^R d_h^T}{p_h}}\right\}\right)$$

We can now set p_h similar to how we chose the values for the Skew Join:

$$p_h = p \cdot \max\left\{\frac{1}{p^{1/3}}, \frac{d_h^R d_h^S}{\sum_{j\in\mathcal{H}_x} d_j^R d_j^S}\right\}$$

This assignment obtains the following load:

$$L = \tilde{O}\left(\max\left\{\frac{m}{p^{2/3}}, \sqrt{\frac{\sum_h d_h^R d_h^S}{p}}\right\}\right)$$

Summing up all the cases, we obtain that the load of the 1-round algorithm for computing triangles is:

$$\tilde{O}\left(\max\left\{\frac{m}{p^{2/3}}, \sqrt{\frac{\sum_h d_h^R d_h^S}{p}}, \sqrt{\frac{\sum_h d_h^R d_h^T}{p}}, \sqrt{\frac{\sum_h d_h^S d_h^T}{p}}\right\}\right)$$

The above algorithm is optimal for computing triangles in one round when the degree distribution is given. Observe that in the case of extreme skew, the load increases from $\tilde{O}(m/p^{2/3})$ in the skew-free case to $\tilde{O}(m/p^{1/3})$.

5. GENERAL JOINS IN ONE ROUND

We have seen so far how to analyze parallel algorithms for computing the join and triangle query. In this section, we generalize our techniques to compute general multiway join queries in *one round*.

5.1 The General HyperCube Algorithm

The algorithm we present here is a generalization of the HyperCube algorithm for triangles. We will consider a multiway join query without projections, called also a *full conjunctive query*:

$$q(x_1, \ldots, x_k) = S_1(\bar{x}_1), \ldots, S_\ell(\bar{x}_\ell)$$

Throughout this section, the size of relation S_j will be m_j. The HC algorithm assigns to each variable x_i, where $i = 1, \ldots, k$, a *share* p_i, such that $\prod_{i=1}^{k} p_i = p$. Each machine is then represented by a distinct point $\mathbf{y} \in \mathcal{P}$, where $\mathcal{P} = [p_1] \times \cdots \times [p_k]$; in other words, machines are mapped into points of a k-dimensional hypercube.

During communication, we use k independently chosen hash functions $h_i : U \to [p_i]$ and send each tuple t of relation $S_j(x_{i_1}, \ldots, x_{i_a})$ to all machines in the destination subcube of t:

$$\mathcal{D}(t) = \{\mathbf{y} \in \mathcal{P} \mid \forall m \in [a] : h_{i_m}(t[i_m]) = \mathbf{y}_{i_m}\}$$

Then, each machine locally computes the query q for the subset of the input that it has received.

The correctness of the HC algorithm follows from the observation that, for every potential output tuple $(a_1, \ldots, a_k)$, machine $(h_1(a_1), \ldots, h_k(a_k))$ contains all the necessary information to decide whether it belongs in the answer or not. Observe also that the choice of $p_1, \ldots, p_k$ gives a different parametrization of the HC algorithm. To analyze the load of the HC algorithm for a particular choice of shares, we will use a generalization of Proposition 3.2.

Definition 5.1. *Let $\mathbf{p} = (p_1, \ldots, p_k)$ be a vector of shares. If for every relation S_j and every tuple of values t over $A \subseteq \bar{x}_j$ the degree of t in S_j is at most $m_j / \prod_{x_i \in A} p_i$, we say that the input is* skew-free *w.r.t.* $\mathbf{p}$.

Proposition 5.2. *Let $\mathbf{p} = (p_1, \ldots, p_k)$ be shares of the HC algorithm. If the input is skew-free w.r.t $\mathbf{p}$, the maximum load is (with high probability)*

$$\tilde{O}\left(\max_j \frac{m_j}{\prod_{i:x_i \in S_j} p_i}\right)$$

The above analysis provides us with a tool to choose the best shares for the HC algorithm. Recall from Section 4 that we can write $p_i = p^{e_i}$, where $e_i \in [0, 1]$ is called the *share exponent* for x_i, and let $\lambda = \log_p L$. Then, we compute the optimal shares by optimizing the following LP:

$$\begin{aligned} \text{minimize} \quad & \lambda \\ \text{subject to} \quad & \sum_{i \in [k]} -e_i \geq -1 \\ & \forall j \in [\ell] : \sum_{x_i \in S_j} e_i + \lambda \geq \log_p(m_j) \\ & \forall i \in [k] : e_i \geq 0, \quad \lambda \geq 0 \end{aligned} \tag{2}$$

Observe that this is the general form of the LP in Section 4. We can now describe our main result on the performance of the HC algorithm. A *fractional edge packing* of a query q is a non-negative weight assignment $\mathbf{u} = (u_1, \ldots, u_j)$ to each relation S_j such that for every variable x_i, $\sum_{j:x_i \in S_j} u_j \leq 1$. Let $\mathsf{pk}(q)$ be the set of all edge packings for q.

Theorem 5.3 (Upper Bound [4]). *Given a query q and p machines, let $\mathbf{e} = (e_1, \ldots, e_k)$ be the optimal solution to (2). Let $p_i = p^{e_i}$, and suppose that the input data is skew-free w.r.t. to $\mathbf{p} = (p_1, \ldots, p_k)$. Then the HC algorithm with shares $\mathbf{p}$ achieves w.h.p.*

$$L = \tilde{O}\left(\max_{\mathbf{u} \in \mathsf{pk}(q)} \left(\frac{\prod_{j=1}^{\ell} m_j^{u_j}}{p}\right)^{1/\sum_j u_j}\right)$$

.

A case of special interest is when all input relations have the same cardinalities, i.e. $m_1 = m_2 = \cdots = m_\ell = m$. In this case the load is $\tilde{O}(m/p^{1/\tau^*})$. Here, τ^* is the fractional edge packing number, defined as $\tau^* = \max_{u \in \mathsf{pk}(q)} \sum_j u_j$.

5.2 Optimality

The HC algorithm is optimal (up to logarithmic factors) among one-round algorithms. Indeed, we can show that there exists a family of skew-free inputs with maximum degree one (called *matching databases*) such that no one-round algorithm can achieve a bound better than the one in Theorem 5.3.

Theorem 5.4 (Lower Bound [5]). *Given a query q, any (randomized) algorithm that computes q correctly in one round with p machines must have maximum load*

$$L = \Omega\left(\max_{\mathbf{u} \in \mathsf{pk}(q)} \left(\frac{\prod_{j=1}^{\ell} m_j^{u_j}}{p}\right)^{1/\sum_j u_j}\right)$$

5.3 Dealing with Skew

If the input data is not skew-free with respect to the optimal share allocation, the HC algorithm does not achieve the optimal anymore. Instead, we

have to use techniques described in the previous two sections to deal with skew, by separating the heavy and light hitters and considering the residual queries. We refer the interested reader to [5] for further details on how to approach general join queries. It still remains an open problem to find a load-optimal one-round algorithm for any input, where optimality in this case is defined with respect to a fixed degree distribution of the input.

6. BEYOND ONE ROUND

In this section, we discuss the analysis of multi-way join algorithms for multiple rounds. Throughout this section, we assume that all input relations have the same size m.

6.1 A Worst-Case Lower Bound

We first present a lower bound for the best possible load of any multi-round algorithm. Atserias, Grohe, and Marx [3] have shown the following result, known as the *AGM bound*: if all input relations have size $\leq m$, then the size of the query is $\leq m^{\rho^*}$, where ρ^* is the *fractional edge covering number* of the query q. Moreover, the AGM bound is tight, in the sense that there exists a "worst case" input database with relation sizes $\leq m$, on which the query returns an answer of size m^{ρ^*}. The fractional edge covering number is defined as the maximum value of $\sum_j u_j$, where the numbers $u_j \geq 0$ are associated to the input relations R_j, and must satisfy the following conditions, for every variable x_i: $\sum_{j:x_i \in R_j} u_j \geq 1$.

THEOREM 6.1 ([17]). *Let q be a join query. Then, there exists a family of instances where relations have the same size m, such that every MPC algorithm that computes q with p machines using a constant number of rounds requires load $\Omega(m/p^{1/\rho^*})$.*

PROOF SKETCH. Assume that an algorithm $\mathcal{A}$ computes q with load L in r rounds. Since each machine receives at most $r \cdot L$ tuples from each relation S_j, we can use the AGM bound to argue that the total number of output tuples will be at most $p(r \cdot L)^{\rho^*}$. If the input database is the worst case input (or close to it), then $\mathcal{A}$ must output m^{ρ^*} tuples and therefore $p(r \cdot L)^{\rho^*} \geq m^{\rho^*}$, or equivalently $L \geq m/(rp^{1/\rho^*})$. □

For the triangle query, since $\rho^* = 3/2$, any algorithm with a constant number of rounds must use load $\Omega(m/p^{2/3})$. Notice that for the triangle query, $\tau^* = \rho^* = 3/2$, and thus we can use the 1-round algorithm from the previous section to compute the query on skew-free data with load $\tilde{O}(m/p^{2/3})$.

Recall that $m/p^{1/\tau^*}$ is the optimal load for one-round algorithms and skew-free data, while $m/p^{1/\rho^*}$ is a lower bound for multi-round algorithms and arbitrary data. In general, there is no relationship between τ^* and ρ^*: for example, the join query has $\tau^* = 1 < \rho^* = 2$, while the query $R(x), S(x,y), T(y)$ has $\tau^* = 2 > \rho^* = 1$.

The upper bound for multi-rounds and arbitrary data is open, except for the case when all input relations are binary: in this case $\tau^* \leq \rho^*$ and it has been shown [16] that the optimal load is precisely $\tilde{O}(m/p^{1/\rho^*})$. Intuitively, after using one round to compute the query on the skew-free data fragment with a load $m/p^{1/\tau^*}$, we can exploit the additional rounds to compute the query on the skewed data values with load $m/p^{1/\rho^*}$. We illustrate this idea next on the triangle query.

6.2 The Triangle Revisited

Recall that the HC algorithm computes Δ on skew-free data with load $\tilde{O}(m/p^{2/3})$ in one round. We will show here that we can compute the query on arbitrary data using the same load in 2 rounds. The main component is a parallel algorithm that computes a semi-join query optimally in a single round, independent of skew.

PROPOSITION 6.2 ([17]). *The semi-join query $q_1(x,y) = R(x), S(x,y)$ can be computed in one round with maximum load $\tilde{O}(m/p)$. Query $q_2(x,y) = R(x), S(x,y), T(y)$ can be computed in two rounds with load $\tilde{O}(m/p)$.*

PROOF SKETCH. To compute q_1 we use Skew Join, outlined in Section 3.4. The data is skewed only on one side, because $R(x)$ is a set; therefore, Skew Join has a load of $\tilde{O}(m/p)$. To compute q_2, in the first round we compute the semi-join $A(x,y) = R(x), S(x,y)$: importantly, the size of the result is no larger than m. In the second round, we compute the semi-join $q(x,y) = A(x,y), T(y)$. Both steps have a load of $\tilde{O}(m/p)$. □

We now describe the 2-round algorithm for computing Δ on arbitrary input data. Recall from Section 4 that a value h is a *heavy hitter* if for some relation the degree of h exceeds $m/p^{1/3}$. For the light values, we can run the HC algorithm in one round and obtain load $\tilde{O}(m/p^{2/3})$.

It remains to output the tuples for which at least one variable has a heavy value. Without loss of generality, consider the case where variable x has heavy values and observe that there are at most $2p^{1/3}$ such heavy values for x ($p^{1/3}$ for R and $p^{1/3}$ for T). For each heavy value h, we assign an *exclusive* set of $p' = p^{2/3}$ servers to compute the query $q[h/x] =$

$R(h, y), S(y, z), T(z, h)$, which is equivalent to computing the residual query $q' = R'(y), S(y, z), T'(z)$. Recall that in the analysis of Section 4, it was expensive to compute this residual query in a single round. However, by Proposition 6.2 using two rounds we can compute each q' with load $\tilde{O}(m/p') = \tilde{O}(m/p^{2/3})$. We have thus shown:

THEOREM 6.3. *The triangle query Δ on input with relation sizes equal to m can be computed by an MPC algorithm in two rounds with $\tilde{O}(m/p^{2/3})$ load, under any input data distribution.*

The 2-round algorithm achieves a better load than any 1-round algorithm in the worst-case scenario. Indeed, there exists an $\Omega(m/p^{1/2})$ lower bound for 1-round algorithms on arbitrary data. By using an additional round, we can beat this bound and achieve a lower load. This confirms our intuition that with more rounds we can reduce the maximum load, even in the case of skewed data.

6.3 General Join Queries

Recall that every algorithm, regardless of the number of rounds, must have load $\Omega(m/p^{1/\rho^*})$. It is currently not known whether this load is optimal for general join queries, but it has been proven to be optimal in [17, 16] for queries where all relations have arity 2.

Our algorithm for computing the triangle query suggests the following general strategy for computing an arbitrary query on arbitrary (possibly skewed) data: (1) compute the query on the light hitters, using one round and load $\tilde{O}(m/p^{1/\tau^*})$. (2) compute the query on the heavy hitters using multiple rounds. If $\tau^* \leq \rho^*$ *and* step (2) can be done with a load of $m/p^{1/\rho^*}$, this algorithm is optimal.

Recall that τ^* and ρ^* are the fractional edge packing number, and the fractional edge covering number of the query respectively. If all relations in the query have arity 2, then the query hypergraph is a graph, and in that case $\tau^* \leq \rho^*$. [17] described an algorithm for step (2) with load $\tilde{O}(m/p^{1/\rho^*})$ for chain queries and for cycles (including the algorithm for the triangle query that we discussed earlier). [16] described a non-trivial extension of this algorithm to arbitrary queries where all relational symbols have arity 2, still having load $\tilde{O}(m/p^{1/\rho^*})$. In all these cases the algorithm consisting of steps (1) and (2) is optimal, since $\tau^* \leq \rho^*$.

It is currently open how to compute optimally queries when $\tau^* > \rho^*$. If the optimal load is indeed $m/p^{1/\rho^*}$, then we need an entirely new approach to compute the query over the light hitters, with a better load than the HC algorithm. To see such an example, consider the query

$$q_3(x_1, x_2, x_3, y_1, y_2, y_3) = R(x_1, x_2, x_3), S_1(x_1, y_1), S_2(x_2, y_2), S_3(x_3, y_3), T(y_1, y_2, y_3).$$

where $\tau^* = 3$ and $\rho^* = 2$. When applied to the light hitters, the HC algorithm allocates equal shares of $p^{1/6}$ to all variables, thus the load is $\tilde{O}(m/p^{1/3})$; this load is also a lower bound for one round algorithms. Yet for multiple rounds the best lower bound is $\Omega(m/p^{1/\rho^*}) = \Omega(m/p^{1/2})$: it is open whether q_3 can be computed with this load in multiple rounds.

7. CONCLUSION

In this paper, we presented recent results on the communication cost of algorithms for computing multiway join queries in modern big data analytics systems. We conclude by discussing some of the key takeaways of our results.

1. The communication cost for 1-round join queries can be reduced by using multi-dimensional hash-partitioning. Each variable x is allocated a share p_x, and the values of x are hash partitioned into p_x buckets. When all relations have the same size m, then the optimal load of any 1-round algorithm is $O(m/p^{1/\tau^*})$, where τ^* is the value of the optimal fractional edge packing.

2. In any partitioning scheme of a relation R, the expected number of tuples received by a machine is the relation size divided by the number of buckets into which it is partitioned. For example, for the triangle query, $R(x, y)$ is partitioned into $p_x p_y = p^{2/3}$ buckets, hence the load is $m/p^{2/3}$.

3. Skew occurs when a data value for some variable x overflows one of its buckets. In a multi-dimensional hash-partitioning scheme, tuples with the same x-value are distributed to fewer buckets than the entire relation, which makes the algorithm quite resilient to skew. For example, in a triangle query a value for variable x is skewed if it occurs more than $m/p^{1/3}$ times. For concrete numbers, assuming $p = 1000$ machines, R is partitioned into 100 buckets, yet we can tolerate values with degree up to $m/10$, much larger than the expected bucket size of $m/100$. There are at most 10 skewed x-values (heavy hitters), regardless of the size of R.

4. Multiple communication rounds can be used effectively to compute queries over skewed data. It is currently open what the optimal load of a multi-round algorithm is. The best lower bound is $\Omega(m/p^{1/\rho^*})$, and this bound is known to be optimal for queries where all relations have arity 2 and the same size.

8. REFERENCES

[1] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4):277–288, 2013.

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

[3] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748, 2008.

[4] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *PODS*, pages 273–284, 2013.

[5] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *PODS*, pages 212–223, 2014.

[6] P. Beame, P. Koutris, and D. Suciu. Communication cost in parallel query processing. *CoRR*, abs/1602.06236, 2016.

[7] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[8] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, pages 63–78, 2015.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[10] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[11] J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina. On distributing symmetric streaming computations. In *SODA*, pages 710–719, 2008.

[12] E. Gribkoff and D. Suciu. Slimshot: In-database probabilistic inference for knowledge bases. *PVLDB*, 9(7):552–563, 2016.

[13] Hadoop. `http://hadoop.apache.org/`.

[14] D. Halperin, V. T. de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the myria big data management service. In *SIGMOD*, pages 881–884, 2014.

[15] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.

[16] B. Ketsman and D. Suciu. A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries, 2017. To appear in PODS.

[17] P. Koutris, P. Beame, and D. Suciu. Worst-case optimal algorithms for parallel query processing. In *ICDT*, pages 8:1–8:18, 2016.

[18] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, pages 223–234, 2011.

[19] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[20] M. Mitzenmacher and E. Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

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

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, pages 1–10, 2010.

[23] Apache spark. `http://spark.apache.org/`.

[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[25] M. S. University and M. Stonebraker. The case for shared nothing. *Database Engineering*, 9:4–9, 1986.

[26] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[27] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Hayes, B. Howe, D. Hutchinson, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whittaker, and S. Xu. The Myria big data management and analytics system and cloud services, 2017. To appear in CIDR.

[28] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.