Parallel Evaluation of Conjunctive Queries*

Paraschos Koutris
University of Washington
Seattle, WA
pkoutris@cs.washington.edu

Dan Suciu
University of Washington
Seattle, WA
suciu@cs.washington.edu

ABSTRACT

The availability of large data centers with tens of thousands of servers has led to the popular adoption of massive parallelism for data analysis on large datasets. Several query languages exist for running queries on massively parallel architectures, some based on the MapReduce infrastructure, others using proprietary implementations. Motivated by this trend, this paper analyzes the parallel complexity of conjunctive queries. We propose a very simple model of parallel computation that captures these architectures, in which the complexity parameter is the number of parallel steps requiring synchronization of all servers. We study the complexity of conjunctive queries and give a complete characterization of the queries which can be computed in one parallel step. These form a strict subset of hierarchical queries, and include *flat* queries like $R(x,y), S(x,z), T(x,v), U(x,w)$, *tall* queries like $R(x), S(x,y), T(x,y,z), U(x,y,z,w)$, and combinations thereof, which we call *tall-flat* queries. We describe an algorithm for computing in parallel any tall-flat query, and prove that any query that is not tall-flat cannot be computed in one step in this model. Finally, we present extensions of our results to queries that are not tall-flat.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Distributed Databases

General Terms

Algorithms, Theory

Keywords

Database Theory, Distributed Databases, Parallel Computation

1. INTRODUCTION

In this paper we study the parallel complexity of conjunctive queries. Our motivation comes from the recent increase in the use of massive parallelism for performing data analysis on very large datasets. In addition to traditional parallel database systems, such as Teradata or Greenplum, new query languages and implementations have been introduced recently for the purpose of massively parallel data analytics: the MapReduce architecture for parallelism [8], SCOPE [4], DryadLINQ [25], Pig [10], Hive [22], Dremel [17]. Most of the effort and the engineering work in these systems has been focused on fault-tolerance, resource allocation, and scheduling, and restricted to basic operations like filtering, group-by, aggregation, and join. As these systems evolve towards general-purpose data analytics languages, they need to optimize and execute in parallel general conjunctive queries.

The parallel complexity of conjunctive queries on today's parallel architectures is not well understood. The data complexity of every relational query is in AC^0 [16], and it is generally acknowledged that SQL is "embarrassingly parallel". Immerman [13] analyzed the parallel complexity of First Order Logic on CRCW PRAM (Concurrent Read, Concurrent Write Parallel Random Access Machine) [21] and showed that the parallel time is equivalent to the number of times one needs to iterate a First Order sentence; an immediate corollary is that *every* relational query takes $O(1)$ parallel time in this model. But circuits and PRAMs are not accurate models of parallel systems. Even in the 80's and 90's researchers have proposed alternative models to capture parallelism. Valiant introduced the BSP model [23] and Culler et al. further refined it into the LogP model [6]. Both view the parallel computation as a sequence of relatively short parallel computation steps, each followed by a global synchronization barrier. The length of a parallel step is called periodicity parameter, L, and most of the analysis of parallel algorithms in these models consists of theoretical guarantees that servers finish their tasks within the periodicity parameter (or else, the next step needs to be dedicated to the unfinished step). Therefore, the models focus on the details of the communication protocol, and trace meticulously the network's latency, overhead, and gap (in the LogP model). These models no longer capture today's parallel architectures well.

Nowadays, massive parallelism is achieved on commodity hardware interconnected by a high speed network. The communication cost is less dependent on the low level protocol, but is dominated by the amount of data being exchanged. In addition, the granularity of a parallel step has increased, since each server needs to process a large amount of data before synchronization, making each synchronization step even more expensive.

The main bottleneck in today's massively parallel computations is the global synchronization steps of a computation [12]. Two factors make a global synchronization step

*This work was partially supported by NSF IIS-0627585 and IIS-0713576.

PODS'11, June 13–15, 2011, Athens, Greece.

expensive: data skew and stragglers. Data skew refers to the fact that some servers end up processing much more data than others. Theoretically, one addresses data skew by requiring that each server is allocated only $O(n/P)$ of the total data, where n is the number of data items and P the number of servers. This guarantees that no server gets an excessive load[1]. Stragglers are a new phenomenon, not encountered in older parallel systems. A straggler is a server that takes significantly longer time to execute its share of the computation than the others. This can be caused by a faulty disk, the server being overloaded with other tasks, by server failure, or by the algorithm itself. Stragglers occur in today's systems because the large number of servers (tens of thousands) and the long processing times (often several hours) dramatically increase the probability that some servers will straggle. All systems to date, starting with the original MapReduce [8], pay special attention to stragglers. They monitor slow responders, and, once a straggler is identified, its work is redistributed to other servers. Despite these measures, stragglers add a significant cost to each synchronization step, and the way to mitigate that cost in a theoretical model is to reduce the number of global synchronization steps.

We propose a simple parallel model of computation, called the *Massively Parallel*, or MP model, to enable us to analyze the parallel complexity of conjunctive queries. In MP, computation proceeds in a sequence of parallel steps, each followed by global synchronization of all servers. We do not impose any restriction on the time of a parallel step. Instead, we impose the restriction that the load at each server is no more than $O(n/P)$ data items during the entire computation, where n is the total size of the input and the output, and P is the number of servers. Each parallel step consists of a broadcast phase (where a limited amount of data is shared among servers, typically in order to detect skewed elements), followed by a communication phase, followed by a computation phase. The cost of an algorithm in the MP model is given by the *number of parallel steps*: we ignore the time of the computation phase and also, in this paper, we ignore the total amount of data transferred during communication. However, notice that the amount of data transferred is always $O(n)$, because the P processors can receive at most $P \cdot O(n/P)$ data items during each communication step; this is in contrast to Afrati and Ullman [1], who allow a total amount of $O(n\sqrt[k]{P^{k-1}})$ data exchanged in one step (see Section 7). Thus, we only count the number of synchronization steps. For example, if Algorithm A computes a query in two parallel steps, each taking time T, and Algorithm B computes the same query in a single parallel step of time $T' = 2T$, then both algorithms take time $2T$, and would be considered equivalent in a traditional model. But, under MP, Algorithm B is better, since it uses one parallel step instead of two.

In this paper we study the evaluation of conjunctive queries in the MP model. We restrict our discussion to *full conjunctive queries* (without projections). Our main result is a complete characterization of queries computable in one MP step; the most significant aspect of this result is that not all queries are easily parallelizable, as older models of parallelism suggest. The queries computable in one MP step are the *tall-flat* queries. A flat query is one where every two atoms share the same sets of variables, for example $q(x, y, z, u) = R(x, y), S(x, z), T(x, u)$ is flat because any two atoms share $\{x\}$; in other words, a flat query is a star join where all join conditions are on the same variable(s). A tall query is one where the set of variables in the atoms forms a linear chain, for example $q(x, y, z, u) = R(x), S(x, y), T(x, y, z), U(x, y, z, u)$; tall queries occur in data warehouse schemas, e.g `Country(`$\underline{co}$`, ...)`, `City(`$\underline{co}, \underline{ci}, \ldots$`)`, `Store(`$\underline{co}, \underline{ci}, \underline{st}, \ldots$`)`. A tall-flat query consists of a tall part followed by a flat part (formal definition in Section 3), and we prove that a query can be evaluated in one MP step iff it is tall-flat. For the "if" part of the proof, we give a concrete algorithm that computes any tall-flat query in one parallel step, while guaranteeing perfect load balance (which is a requirement in the MP model), even if the data is skewed. We give the algorithm in two stages: we describe the algorithm separately for tall, and for flat queries in Section 4, then combine them into a general algorithm in Section 5. The simple case of a 2-way join $q(x, y, z) = R(x, y), S(y, z)$ can be computed by an alogorithm similar in spirit to the *skew join* in Pig [10]; k-way flat queries with $k \geq 3$ and tall queries require non-trivial extensions. For the "only if" part of the proof, we show in Section 6 that for any non tall-flat query, any one-step parallel algorithm will be skewed, thus violating the MP model.

Our results depend critically on the load balance requirement, which strictly limits the amount of data per processor to $O(n/P)$, and, hence, limits the communication cost to $O(n)$. Afrati and Ullman [1] describe a simple algorithm for computing the query $RST(x, y) = R(x), S(x, y), T(y)$ (which is not tall-flat) in one parallel step, by allowing each server to store $O(n/\sqrt{P})$ amount of data, and, thus, with communication cost $O(n\sqrt{P})$. In general, any conjunctive query with k variables can be computed in one parallel step if one allows the load per processor to increase to $O(n/\sqrt[k]{P})$ (and the communication cost to $O(n\sqrt[k]{P^{k-1}})$).

Organization. We first discuss in Section 2 about related work in parallel and distributed database models. Then, we describe the model and the main algorithmic techniques in Section 3, and the main algorithms in Section 4. We prove the main result in Section 5 and Section 6. Finally, we discuss some extensions in Section 7 and conclude in Section 8.

2. RELATED WORK

The recent success of the MapReduce framework [8] in efficiently parallelizing computations has inspired theoretical research on new models of parallel computation, which capture the characteristics and limitations of the new architecture, while also considering the capabilities of modern hardware and infrastructure.

Afrati and Ullman [1] describe a model of computation where every query is executed in one parallel step and the primary measure of complexity is communication. Our model always restricts the communication to $O(n)$, but needs multiple communication steps, hence this is our main complexity metric. By contrast, in their work they allow a larger amount of communication, typically $O(n\sqrt[k]{P^{k-1}})$, and therefore the main complexity metric is the total amount of communication.

Another theoretical approach to analyzing MapReduce is presented in [14]. In this paper, the authors allow only a limited amount of storage in each processor, and add randomization, in the sense of allowing false computations as well. Their main measure of complexity is similar to ours: the number of MapReduce steps; however, they do not examine queries, but general algorithms, comparing their model to the PRAM model.

[1]We assume that each of the n items takes about the same time to process. This assumption may fail for jobs with user defined functions.

Measuring parallel complexity in terms of the the number of communication steps was also proposed by Hellerstein [12], who introduced the notion of *Coordination Complexity*. The author argues that both communication and computation are cheap using today's infrastructure; the bottleneck in parallelizing queries lies in the coordination of global barriers during computation.

Apart from recent approaches to modeling the MapReduce framework, various parallel models have been extensively studied, for example the widely used *bulk-synchronous parallel* model [23] and the LogP model [6]. However, both models do not seem to adequately capture the concerns of today's massively parallel computations.

The idea that we handle skewed elements in a special way during computation is not a new one; in [24], the authors develop and implement an algorithm similar to ours that handles skewed join computation in shared-nothing architectures. Nevertheless, they assume that they have knowledge of the skewed elements and do not provide any theoretical guarantees on their algorithm.

Grohe et al. [11] introduced the *Finite Cursor Machines* model of computation, in which queries are evaluated in a sequence of "streaming" steps: each relation can be sorted at the beginning of the step, but afterwards can be read a constant number of times using a constant amount of memory, in a streaming fashion. They do not restrict to full queries, but they do restrict to conjunctive queries in the *semi-join algebra* in order to ensure that the output is linear in the size of the input (this rules out cartesian product queries like $q(x, y) = R(x), S(y)$). For example, a query like $q(x) = R(x, y), S(x, z), T(x, u)$ can be computed in one step, by first sorting the relations on x, then merge-joining the three streams. The authors prove that the query $q(x, y) = R(x), S(x, y), T(y)$ cannot be computed in one streaming step in this model. A conjunctive query is called *hierarchical* if, denoting $at(x)$ the set of atoms that contain the variable x, the family of sets $\{at(x) \mid x \in Var(Q)\}$ form a hierarchy (formal definition in Subsection 3.1). Although the authors [11] do not state this explicitly, it follows immediately from their result that a conjunctive query in the semi-join algebra can be evaluated in one step iff it is hierarchical. Dalvi et al.[7] show that, over tuple-independent probabilistic databases, queries that can be evaluated efficiently are precisely the hierarchical queries [2]. Thus, both in the streaming and probabilistic data model, the tractable queries have the property of being hierarchical. In our work we consider full conjunctive queries, but do not restrict to the semi-join algebra. Thus, we allow cartesian product queries, and show that the queries computable in one pass are the tall-flat queries: for example the cartesian product query $q(x, y) = R(x), S(y)$ is flat, and therefore computable in one MP step. Yet, not every hierarchical query is tall-flat, for example $q(x, y) = R(x), S(x), T(y)$, and these cannot be computed in one MP step (Section 6). When restricting to queries that are both full *and* in the semi-join algebra of [11], we prove that hierarchical queries coincide with tall queries and tall-flat queries collapse to tall queries [15]. Thus, for the intersection of the language considered in [11] and the one in this paper, the class of "easy" queries coincides.

3. THE MODEL AND THE MAIN RESULT

We define here the *Massively Parallel* (MP) model of query evaluation. We fix throughout this paper a domain, U, called *universe*. A relational database instance D contains constants from U, and has a relational schema $\mathbf{R} = R, S, T, \ldots$ We consider in this paper only *full* conjunctive queries, i.e. where all variables are head variables. Denote the *problem size* to be the size of the input database plus the size of the query's answer, $n = size(D) + size(Q)$.

Let P be the number of parallel servers. Each server stores two kinds of data: *generic data* (values from U) and *numerical data* (integers). The data is stored in arrays, on disks or in main memory. Generic values can be manipulated only in three ways: they can be copied, they can be tested for equality $a = b$, and they can be subjected to a hash function, from a fixed set of hash functions $\bar{h}$. Each hash function has type $h : U^k \to [P]$, where k is its arity. For example, an algorithm may use three hash functions, $\bar{h} = (h_1, h_2, h_3)$, where $h_1, h_2 : U \to [P]$ and $h_3 : U^3 \to [P]$. Each hash function is randomized at the beginning of the algorithm, i.e. chosen randomly from a finite set[3] $\mathcal{H}$ of a family of hash functions [4].

Initially, all relations are already uniformly distributed over the servers (so no additional communication is necessary), and their sizes are known by all servers.

An algorithm in the MP model runs as follows. The *input* is the database instance D. Each relation R is partitioned into P fragments $R_1, R_2, \ldots, R_P$ of equal size, and distributed over the P servers; server s holds the fragments $R_s, S_s, T_s, \ldots$ The initial load of each server is $size(D)/P$. The algorithm proceeds in a number of *parallel computation steps*, each consisting of three phases:

Broadcast Phase: The P servers exchange some data globally. This data is shared among all servers, and we call it *broadcast data*, B. At the end of this phase each server has a copy of B. We require the total amount of communication in this phase to be $O(n^\varepsilon)$, for some $0 < \varepsilon < 1$; in particular, $size(B) = O(n^\varepsilon/P)$.

Communication Phase: Each server sends data to other servers. There is no restriction imposed on how a server distributes the data to the other servers; it could send its entire data to a single server, distribute it uniformly among servers, or broadcast its entire data to all servers (but see the skew-free requirement below).

Computation Phase: Each server performs some local computation on its data. There is no restriction imposed on the time taken for the computation. At the end of this phase the algorithm may stop and leave the result in the local memory of the P servers, or may continue with the next parallel step.

We allow local computation to be performed during all three phases; in our model local computation is free. A *parallel algorithm* is a sequence of parallel computation steps, and its *cost* is the number of parallel steps. Let n_s denote the total number of tuples stored at sever s during the entire execution of the algorithm. The algorithm is called *load balanced*, or *skew free*, if, for sufficiently large n, $\mathbb{E}[\max_s n_s] = O(n/P)$, where the expectation is taken over the random choices of the hash functions $\bar{h} \in \mathcal{H}$. Formally:

[2]This result applies only to conjunctive queries without self-joins.

[3]Strictly speaking, we need a separate family for each hash function used: $h_1 \in \mathcal{H}_1$, $h_2 \in \mathcal{H}_2$, etc. To simplify things, we use a single family and write $\bar{h} \in \mathcal{H}$ with some abuse.

[4]Although in this paper we allow randomization only in the choice of hash functions (with the exception of Subsection 4.1), all the hardness results (apart from Proposition 4.3, which we defer to the full version) easily extend to arbitrary use of randomization.

DEFINITION 3.1. *An algorithm is* load balanced *if there exists a constant factor $c > 0$ s.t. for all number of servers P there exists an integer n_0 such that size$(D) = n \geq n_0$, implies $\mathbb{E}_{\bar{h}\in\mathcal{H}}[\max_{s=1,P} n_s] \leq c \cdot n/P$.*

We require every algorithm to be load balanced. This requirement is a key element of the MP model. Without it, any query could be computed in one step, because all servers could send their data to server number 1, which computes the query locally. It has two consequences. First, it ensures linear speedup [9] (by doubling the number of servers P, the load per server will be cut in half[5]) and constant scaleup (by doubling both the size of the data n and the number of processors P, the running time remains unchanged). Second, it implies that the total amount of data exchanged during each communication step is $O(n)$ in expectation. Indeed, let n_s be the number of tuples received by server $s = 1, P$: since $\mathbb{E}[n_s] = O(n/P)$, the total amount of data exchanged is $\mathbb{E}[\sum_s n_s] = O(n)$.

Thus, the amount of communication is guaranteed to be $O(n)$. In most algorithms, this is also a lower bound, but in some cases one can design algorithms with strictly less communication. For example, consider the intersection query $q(x) = R(x), S(x)$; *partitioned hash join* has communication cost $n = |R| + |S|$ (Proposition 3.3). But if we know that $|R| \ll |S|$, then we can use *broadcast join*: broadcast R to all servers, then each server intersects R with its local fragment of S. The communication cost is $P \cdot |R|$, which may be much less than n. In this paper, we do not distinguish between these two algorithms, since both require one parallel step.

The MP model is related to, but not identical to a Map Reduce (MR) job [8]. MR corresponds to one MP step: there is no broadcast phase, the communication phase is the *map job* followed by data reshuffling, and the computation phase is the *reduce job*. However, to implement an MP algorithm over MR one has to implement the broadcast phase either as a separate, lightweight MR job, or by sampling the data before it is partitioned into servers at the beginning of the map job. As we show in Proposition 3.5, the broadcast phase is necessary to ensure load balance even for the simplest of the join algorithms, and this is why we include it in the model. Using sampling in order to improve load balance is a popular technique in practice, for example it is used in *skew join* in Pig [10].

Finally, note that in Definition 3.1, n_0 is allowed to depend on P. In other words, once P is fixed, load balance is expected only for n "large enough". In some of our algorithms we require $P = O(n^{\varepsilon})$, for some $0 < \varepsilon < 1$.

3.1 Main Result

The main result in this paper is a complete characterization of queries that are computable in one MP step by a load balanced algorithm. To describe this result we first need some definitions.

Given a conjunctive query Q and a variable x, denote by $at(x)$ the set of atoms that contain x. A query is called *hierarchical* if for any two variables x, y, one of the following holds: $at(x) \subseteq at(y)$ or $at(x) \supseteq at(y)$ or $at(x) \cap at(y) = \emptyset$.

A conjunctive query is called *tall-flat* if one can order its variables $x_1, \ldots, x_k$, $y_1, \ldots, y_\ell$ such that: (1) $at(x_1) \supseteq at(x_2) \supseteq \ldots \supseteq at(x_k)$, (2) $at(x_k) \supseteq at(y_i)$ for $i = 1, \ldots, \ell$, and (3) $|at(y_i)| = 1$. Clearly, every tall-flat query is hierarchical. Furthermore, if $l = 0$ then we call it a tall query, and if $k = 1$ then we call it a flat query. For example the query

$$L(x_1, x_2, x_3, x_4, y_1, y_2, y_3) : -$$
$$R_1(x_1), R_2(x_1, x_2), R_3(x_1, x_2, x_3),$$
$$R_4(x_1, x_2, x_3, x_4), S_1(x_1, x_2, x_3, x_4, y_1),$$
$$S_2(x_1, x_2, x_3, x_4, y_2), S_3(x_1, x_2, x_3, x_4, y_3)$$

is a tall-flat query (Figure 1). The main result we prove is:

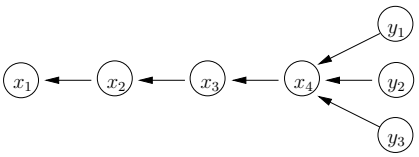

Figure 1: The tall-flat query L. An arrow $u \rightarrow v$ denotes that $at(u) \subseteq at(v)$.

THEOREM 3.2. *Every tall-flat conjunctive query can be evaluated in one MP step by a load balanced algorithm. Conversely, if a query is not tall-flat, then every algorithm consisting of one MP step is not load balanced.*

3.2 Datalog Notation for MP Algorithms

Throughout the paper, we express algorithms using a simple extension to non-recursive datalog, which allows us to specify the location where data is stored. For this purpose we adopt the syntax from [2]. If $R(x, y)$ is a relation (binary in this case), then the notation `R(@s,x,y)` denotes the *fragment* of the relation R which is stored at server s. Using this notation, we can define computations and communication in the following way:

- Local computation: `R(@s,x,y):-S(@s,x,y),T(@s,x)`
- Broadcast: `R(@*,x):-S(@s,x),T(@s,x)`
- Point-to-point communication: `R(@h(x),x,y):-S(@s,x,y)`

3.3 Examples

We illustrate our model by giving two simple algorithms, for computing the intersection of two sets, and a semijoin.

The first query computes the intersection of two sets R, S, $Q(x) : -R(x), S(x)$. This query is both tall and flat, and can be computed by the distributed hash-join Algorithm 1.

Algorithm 1: INTERSECTION$(R(x), S(x))$

```
/* Communication Phase */
R2(@h(x), x) :- R(@s, x)
S2(@h(x), x) :- S(@s, x)

/* Computation Phase */
Q(@s, x) :- R2(@s, x), S2(@s, x)
```

In this algorithm, h is a hash function $h : U \rightarrow [P]$. Each server s applies the hash function h to each tuple $R(a)$ and sends it to the destination server $h(a)$; similarly for tuples $S(b)$. In the computation phase, the tuples placed in the same server are joined. Clearly, this algorithm is correct, and runs in one parallel step, having no broadcast phase. We will prove next that it is load balanced.

No fixed hash function h can guarantee load balance, because there exists a worst case data instance such that all its data values collude under h. Instead, we follow here the

[5]And, thus, the time of the computation phase will also be halved, assuming that the computation time is linear in the size of the data.

standard approach, and assume that h is a *uniform family*[6] of hash functions [18]. Thus, in general, any MP algorithm starts by choosing randomly its hash functions from $\mathcal{H}$: once these choices $\bar{h} \in \mathcal{H}$ are made, we call the execution of the algorithm a *run*. The load balance requirement can be rephrased as follows: $\mathbb{E}_{\bar{h}\in\mathcal{H}}[\max_s n_s] = O(n/P)$, where the expectation is taken over all runs.

PROPOSITION 3.3. *Assuming $n = \Omega(P \log P)$, Algorithm 1 is load balanced, i.e. the expected maximum server load is $\mathbb{E}[\max_s n_s] = O(n/P)$.*

PROOF. Consider the classical *balls in bins* problem: n balls are randomly thrown into P bins. If $n = \Omega(P \log P)$, the expected maximum load of a bin is $\Theta(n/P)$ [19]. This immediately proves the claim: each of the n tuples in R and S is a ball, and h places each tuple independently in one of the P servers (= bins), because h is chosen from a uniform family of hash functions, hence $\mathbb{E}[\max_s n_s] = \Theta(n/P)$. □

Our second example illustrates the broadcast phase. Consider the query $Q(x, y) : -R(x), S(x, y)$: this is a semijoin, and it is a tall query. A naive extension of the previous algorithm would partition both $R(x)$ and $S(x, y)$ according to a hash function $h(x)$, but this may result in data skew: for example, if all tuples $S(x, y)$ have the same value $x = a$, then they will be hashed to the same server $h(a)$, whose load increases to $O(n)$. Here it is necessary to obtain some information about the distribution of tuples in S, using the broadcast phase. Algorithm 2 performs a load balanced computation of the semijoin query.

Algorithm 2: SEMI-JOIN$(R(x), S(x, y))$

```
/* Broadcast Phase */
/* Compute skewed elements; τ = |S|/(P log P) */
 G(@s, x, count(*)) :- S(@s,x,y)
 H(@s, x) :- G(@s,x,f), f > τ/P
 B(@*, x) :- H(@s, x)    /* Broadcast */
/* Communication Phase */
 R2(@h(x), x) :- R(@s, x), not B(@s, x)
 S(@h(x), x, y) :- S(@s, x, y), not B(@s, x)
 R2(@*, x) :- R(@s, x), B(@s, x)
 S(@h2(x,y), x) :- S(@s, x, y), B(@s, x)
/* Computation Phase */
 Q(@s, x, y) :- R2(@s, x), S2(@s, x, y)
```

In general, for any relation $S(x, \ldots)$ and attribute x of S, define the *frequency* $f_{S,x}(a)$ of a constant a to be the number of elements in S that have a on the x position. Let $\tau > 0$ be a value called a *threshold*. A value a is called (S, x, τ)-skewed, or simply *skewed*, or *frequent*, if $f_{S,x}(a) \geq \tau$; we denote $F_{S,x,\tau}(x)$ the set of skewed elements. Notice that $|F_{S,x,\tau}| \leq |S|/\tau$.

The semi-join Algorithm 2 starts by computing all τ-skewed elements, for $\tau = |S|/(P \log P)$. This set cannot be efficiently computed exactly, because S is distributed; instead we compute a superset B. Each server s computes all elements x whose local frequency is $\geq \tau/P$; the `count(*)` notation is from [5]. There are $\leq (|S|/P)/(\tau/P) = |S|/\tau$ locally skewed elements. These sets are broadcast, and each server computes their union, B. The set B contains $F_{S,x,\tau}$, and $|B| \leq P \cdot |S|/\tau$; thus the communication cost of the broadcast phase is $\leq P^2 \cdot |S|/\tau = P^3 \log P = O(n^\varepsilon)$ (we show in Subsection 4.1 how to further improve this). Next, the semi-join algorithm proceeds as follows. For all non-skewed elements x, both $R(x)$ and $S(x, y)$ are sent to the server $h(x)$; for the skewed elements, $S(x, y)$ is distributed using a second hash function $h2(x, y)$, while $R(x)$ is broadcast to all servers. This is similar to *skew join* in Pig [10], but computes the skewed elements differently.

PROPOSITION 3.4. *Assuming $n = \Omega(P^3 \log P)$, Algorithm 2 is load balanced.*

The proof follows as a corollary of Theorem 4.4, which we prove in the next section.

We now prove that, without a broadcast phase, no load-balanced algorithm can compute the semi-join in one step. This justifies including a broadcast phase in the MP model.

PROPOSITION 3.5. *If an algorithm for computing the semi-join $Q(x, y) : -R(x), S(x, y)$ has one single parallel step and no broadcast phase, then it is skewed.*

PROOF. We start by revisiting how the data is partitioned. In the MP model the input data is partitioned *jointly*: each sever holds a fragment from each relation. But in impossibility proofs we will assume that the database is partitioned *separately*: each server holds a fragment from only one relation: $P \cdot |R|/(|R| + |S|)$ severs hold fragments of R, and $P \cdot |S|/(|R| + |S|)$ servers hold fragments of S. Any impossibility result for the separate partition implies an impossibility result for the joint partition, because any load balanced algorithm A over the separate partition can be simulated by a balanced algorithm A' over the joint partition as follows. Given a separately partitioned instance R, S, extend it to a joint partitioned instance R', S', by inserting new R-tuples in the servers holding S, and inserting new S-tuples in the servers holding R, such that $|R'| = |S'| = |R| + |S|$. The new tuples are chosen s.t. they do not join with any other tuples. Run A' on R', S': the result is $S' \ltimes R' = S \ltimes R$. Thus, w.l.o.g., we will assume a separate partition in all impossibility proofs.

Assume A is a load balanced algorithm computing the query Q. Let c be the constant in Definition 3.1, let $P \geq 64c^2$, and let $n = n_0$, where n_0 is given by Definition 3.1 (it may depend on P). We will show that A does not compute correctly Q on an instance of size $2n$ using P processors. To simplify the notations we assume that A uses a single hash function h, of arity k: the extension to multiple hash functions is straightforward. Fix two n-vectors $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_n) \in U^n$ of distinct elements. For each $m = 1, n$, denote $D^{(m)} = (R, S^{(m)})$ the instance $R = \{x_1, \ldots, x_n\}$, and $S^{(m)} = \{(x_m, y_1), \ldots, (x_m, y_n)\}$. For any run $h \in \mathcal{H}$, let $K_h(R(x_i))$ and $K_h(S(x_i, y_j)) \subseteq [P]$ be the set of servers that receive $R(x_i)$ or $S(x_i, y_j)$ respectively.

LEMMA 3.6. *If A is load balanced, then for any $X, Y \in U^n$, there exists an instance $D^{(m)}$ (constructed from X, Y, m as explained above), a run $h \in \mathcal{H}$, and an element $y_i \in Y$, such that $K_h(R(x_m)) \cap K_h(S(x_m, y_i)) = \emptyset$.*

PROOF. Let $n_s(D^{(m)}, h)$ be the load at server s for the instance $D^{(m)}$ and the run h. Since $size(D^{(m)}) = 2n$, by Definition 3.1, $\mathbb{E}_{h\in\mathcal{H}}(\max_s n_s(D^{(m)}, h)) \leq 2 \cdot c \cdot n/P$. Let $d = 4 \cdot c$. We say that h is *balanced for* $D^{(m)}$ if $\max_s(n_s(D^{(m)}, h)) \leq 2 \cdot d \cdot n/P$, and we say that h is *balanced* if it is balanced for some $D^{(m)}$. Denote by $\mathcal{H}^{(m)}$ the set of runs balanced

[6]The family $\mathcal{H}$ is called *uniform* on a set $S = \{x_1, \ldots, x_k\}$ if $h \in \mathcal{H}$ maps S to k values that are uniformly random and independent. Uniformity is strictly stronger than *universality* [3]. In all our algorithms, $|S| = O(n)$.

for $D^{(m)}$, and $\mathcal{H}^{(*)} = \bigcup_m \mathcal{H}^{(m)}$. By Markov's inequality, $|\mathcal{H}^{(m)}|/|\mathcal{H}| \geq 1 - c/d = 3/4$, and also $|\mathcal{H}^{(*)}|/|\mathcal{H}| \geq 3/4$: thus, most runs are balanced.

Call an element x_i *good* for run h if, on some input $D^{(m)}$, $|K_h(R(x_i))| < P/(2 \cdot d)$: "goodness" does not depend on $D^{(m)}$, because the input is partitioned separately, and all instances $D^{(m)}$ have the same R. We claim that there exists m such that x_m is good for some $h \in \mathcal{H}^{(m)}$. The claim proves the lemma: indeed, consider the run h on the instance $D^{(m)}$. Each server in $K_h(R(x_m))$ stores at most $2 \cdot d \cdot n/P$ tuples (since h is balanced for $D^{(m)}$); hence, together they hold $< P/(2 \cdot d) \times 2 \cdot d \cdot n/P = n$ tuples. This implies that at least one of the n tuples $S(x_m, y_i)$, $i = 1, n$ is not sent to any server in $K_h(R(x_m))$.

To prove the claim, denote for each $h \in \mathcal{H}^{(*)}$:

$$B_h = \{x_i \mid |K_h(R(x_i))| \geq P/(2 \cdot d)\} \quad (\textit{bad} \text{ elements})$$

$$G_h = \{x_i \mid |K_h(R(x_i))| < P/(2 \cdot d)\} \quad (\textit{good} \text{ elements})$$

Clearly, $B_h \cup G_h = \{x_1, \ldots, x_n\}$. Since h is balanced for some $D^{(m)}$ we have $|B_h| \cdot P/(2 \cdot d) \leq 2 \cdot d \cdot n$. Therefore:

$$|G_h| \geq (1 - \frac{4d^2}{P}) \times n \geq 3/4 \cdot n \qquad \forall h \in \mathcal{H}^{(*)} \qquad (1)$$

The last inequality holds because $P \geq 64c^2 = 16d^2$. We now prove that there exists a set $\mathcal{H}' \subseteq \mathcal{H}^{(*)}$ such that:

$$\bigcap_{h \in \mathcal{H}'} G_h \neq \emptyset \quad \text{and} \quad |\mathcal{H}'| \geq 3/4 \cdot |\mathcal{H}^{(*)}| \geq 9/16 \cdot |\mathcal{H}| \quad (2)$$

For that, consider the $\{0,1\}$-matrix $E = (e_{ih})$ of dimensions $n \times |\mathcal{H}^{(*)}|$, where $e_{ih} = 1$ iff $x_i \in G_h$. By Equation 1, every column h has at least 3/4 of entries equal to 1; thus, at least 3/4 of entries in the entire matrix are 1; thus, there exists a row m with at least 3/4 of the entries 1. Then, $\mathcal{H}' = \{h \mid e_{mh} = 1\}$ satisfies Equation 2.

To prove the claim, we show that $\mathcal{H}^{(m)} \cap \mathcal{H}' \neq \emptyset$. This follows from $|\mathcal{H}^{(m)}| + |\mathcal{H}'| \geq 3/4 \cdot |\mathcal{H}| + 9/16 \cdot |\mathcal{H}| = 21/16 \cdot |\mathcal{H}|$: the two sets combined have more elements than $\mathcal{H}$ and therefore have a non-empty intersection. □

Fix $D = D^{(m)} = (R, S^{(m)})$, and the run h given by the lemma. We will show that the algorithm is incorrect on the run h, by showing that it fails to output the tuple (x_m, y_i). Servers $\notin K_h(S(x_m, y_i))$ cannot output this tuple, since they don't receive y_i and cannot fabricate a generic value in U. Servers $\in K_h(S(x_m, y_i))$ don't receive $R(x_m)$. To show that they cannot output this tuple either, consider the execution of the algorithm on a second instance $D' = (R', S^{(m)})$ where R' is obtained from R by replacing the single tuple $R(x_m)$ with $R(x'_m)$, where $x'_m \neq x_m$: (x'_m, y_i) is no longer an answer to Q on D'. Servers holding input fragments of S will send their data to the same destinations, since S is unchanged; in particular, $S(x_m, y_i)$ is sent to the same set $K_h(S(x_m, y_i))$. We claim that, during the two execution of the algorithm, on D and D' respectively, the server holding $R(x_m)$ (in D) or $R(x'_m)$ (in D') will obtain exactly the same hash values; in particular, it will send its elements to exactly the same destinations, hence $K_h(R(x'_m)) \cap K_h(S(x_m, y_i)) = \emptyset$. Thus, the servers in $K_h(S(x_m, y_i))$ receive exactly the same elements during the two executions, and will not be able to distinguish between D and D', thus cannot determine whether to output the tuple (x_m, y_i).

It remains to prove the claim. For $X \in U^n$ denote $\mathcal{H}(X)$ the array consisting of all hash functions $h \in \mathcal{H}$ applied to all values in X. One can view $\mathcal{H}$ as assigning a color to each point in U^n. The number of colors is finite, c: for example, if all hash functions in $\mathcal{H}$ have type $h : U^k \to [P]$, then $c = [P]^{|\mathcal{H}| \cdot n^k}$. Call a set $\bar{V} = V_1 \times \cdots \times V_n \subseteq U^n$ a *p-space* if $|V_i| \geq p$, for $i = 1, n$. A p-space is *unicolored* if all its points have the same color. We prove that there exists a unicolored 2-space $\{x_1, x'_1\} \times \cdots \times \{x_n, x'_n\}$. This implies Corollary 3.8, which proves the claim.

LEMMA 3.7. *Suppose all points in U^n are colored with c colors. Then, for every $p > 0$ there exists $M > 0$ such that every M-space has a unicolored p-subspace. In particular, if U is infinite, then U^n has a unicolored 2-subspace.*

PROOF. We set $M = f_n(p)$, where $f_1(p) = (p-1) \cdot c + 1$, $f_{n+1}(p) = f_n^{(f_1(p))}(p)$ (where $f_n^{(k)}(p) = f_n(\cdots f_n(p) \cdots)$, k times). The proof is by induction on n. For $n = 1$, consider a set V_1 with $f_1(p)$ points: at least p have the same color. Assuming the lemma is true for n, we prove it for $n+1$. Let $M = f_{n+1}(p)$ and consider an $(n+1)$-dimensional M-space $\bar{V}^0 \times V_{n+1}$, where $\bar{V}^0 = V_1 \times \cdots \times V_n$. Let $k = (p-1) \cdot c + 1$. Fix k distinct elements $x_1, \ldots, x_k \in V_{n+1}$ (this is possible since $|V_{n+1}| \geq M = f_{n+1}(p) \geq f_1(p) = k$). Denote $p_k = p$, $p_{i-1} = f_n(p_i)$ for $i = k, \ldots, 1$; thus, $M = p_0$. For every $i = 1, k$ there exists an n-dimensional p_i-space $\bar{V}^i$ such that $\bar{V}^i \times \{x_i\}$ is unicolored, and $\bar{V}^i \subseteq \bar{V}^{i-1}$: indeed, $\bar{V}^0 \times \{x_1\}$ is an n-dimensional space, hence by induction on n has a unicolored subspace $\bar{V}^1 \times \{x_1\}$; suppose $\bar{V}^{i-1} \times \{x_{i-1}\}$ is unicolored: since $\bar{V}^{i-1} \times \{x_i\}$ is an n-dimensional p_{i-1}-space, it has a unicolored p_i-subspace $\bar{V}^i \times \{x_i\}$. Thus, we obtain k unicolored spaces, $\bar{V}^1 \times \{x_1\}$, …, $\bar{V}^k \times \{x_k\}$, and at least p of them have the same color: $\bar{V}^{i_1} \times \{x_{i_1}\}$, …, $\bar{V}^{i_p} \times \{x_{i_p}\}$. Then $\bar{V}^{i_p} \times \{x_{i_1}, \ldots, x_{i_p}\}$ is a unicolored p-space. □

COROLLARY 3.8. *There exist vectors $X = (x_1, \ldots, x_n)$ and $Y = (x'_1, \ldots, x'_n)$ such that for any m, the vectors X and $X' = (x_1, \ldots, x'_m, \ldots, x_n)$ collude.*

4. THREE BUILDING BLOCKS

In this section, we describe the building blocks for the algorithm computing any tall-flat query in one MP step: we give an algorithm for computing the frequent elements of a relation, an algorithm for computing any flat query, and an algorithm for computing any tall query. In Section 5 we combine them to give a general algorithm for any tall-flat query.

4.1 Computing High Frequency Elements

The *distributed frequent elements problem* is the following: given a distributed relation $R(x, \ldots)$ of size $r = |R|$, and a threshold τ, compute a set F containing all skewed elements $F_{R,x,\tau}$, and distribute F to all servers. We consider algorithms that proceed in two steps: (a) compute and broadcast a set B to all servers, and (b) compute a subset F of B at each server s.t. F is a superset of $F_{R,x,\tau}$. We define two costs: the *amortized communication cost*, $|B|$, and the *excess cost*, $|F|$. Our goal is to keep the amortized cost $O(n^\varepsilon/P)$, becuase the total communication cost is $P \cdot |B|$. The excess is at least r/τ, because, in the worst case, there are r/τ frequent elements $F_{R,x,\tau}$: our goal is to prevent it from being much larger. We present here three algorithms for the distributed frequent element problem.

The *naive algorithm* consists of the top three rules in Algorithm 2. Here $F = B$, hence both amortized cost and excess are $\leq P \cdot r/\tau$.

The *Deterministic Frequency*, Algorithm 3, has essentially the same amortized cost, but a smaller excess. It starts by computing all elements whose local frequency is $\geq \tau/(2P)$, and retains their local frequencies, then broadcasts these elements: thus, the amortized communication cost is $|B| \leq$

Algorithm 3: DETERMINISTIC FREQUENCY(R, τ)

```
HS(@s,x,count(*)):- R(@s,x,...)
G(@s,x,f) :- HS(@s,x,f), f > τ/(2P)
B(@*,x,f,s) :- G(@s,x,f) /* Broadcast */
H(@s,x,sum(*)) :- B(@s,x,f,-)
F(@s,x) :- H(@s, x, f), f ≥ τ/2
```

$2 \cdot P \cdot r/\tau$, twice that of the naive algorithm. Then, it retains in F only those elements x whose total frequency exceeds $\tau/2$; therefore, the excess is $|F| \leq 2 \cdot r/\tau$, a factor of $P/2$ smaller than the naive algorithm.

Algorithm 4: FREQUENCY SAMPLING(R, τ)

```
T = c · r · log r/τ /* c is a constant */
t_s ~ B(r, T/(P · r)) /* binomial sample */
H(@s, x, ...)  :- sampling(@s, R, t_s)
G(@s, x, count(*)) :- H(@s, x, ...)
B(@*, x, s) :- H(@s, x, ...) /* Broadcast */
K(@s, x, count(*)) :- B(@s, x, -)
F(@s, x) :- K(@s, x, f), f ≥ c · τ · T/r
```

The *Frequency Sampling*, Algorithm 4, has both a smaller amortized communication, and a smaller excess cost over the naive algorithm. Denote $T = c \cdot r \cdot \log r/\tau$, where $c > 0$ is some constant. The algorithm starts by computing a "coin-flip" sample B of R, where each element of R is sampled independently, without replacement, with probability T/r. Thus, $\mathbb{E}[|B|] = T$, and the amortized communication cost is $c \cdot r \cdot \log r/\tau$ in expectation, which is significantly lower than $P \cdot r/\tau$ when $\log r \ll P$. To compute B distributively, each server s generates a random number t_s drawn from a binomial distribution $B(r, T/(P \cdot r))$ (thus, $\mathbb{E}[t_s] = T/P$), then samples t_s elements without replacement from its local fragment R_s, using the primitive function `sampling`$(@s, R, t_s)$. Once B has been computed and broadcast, we retain in F only those elements whose total frequency is $\geq c \cdot \tau \cdot \frac{T}{r}$. We prove in the appendix (Proposition A.1) that, with high probability, the frequency of all elements in F is $\Omega(\tau)$, hence the excess is, in expectation, $O(r/\tau)$.

4.2 Flat Queries

We start with Algorithm 5, which computes the full join $Q(x, y, z) : -R(x, y), S(x, z)$. Similarly to the semijoin algorithm, during the broadcast phase we compute the high frequency elements in R and in S. We set frequency threshold for R to $\tau_R = \sqrt{r/(P \log P)}$ and for S to $\tau_S = \sqrt{s/(P \log P)}$. Using one of the distributed high frequency algorithms in Subsection 4.1, we compute a set RF containing all frequent elements F_{R,x,τ_R} and a set SF containing all frequent elements F_{S,x,τ_S}. Then we proceed similarly to the semijoin algorithm: if x is frequent in R then we duplicate the $S(x, z)$ elements; otherwise, if it is frequent in S then we duplicate the $R(x, y)$ elements; otherwise we don't duplicate, but hash on x.

THEOREM 4.1. *Assuming $|R|/\log^2|R| = \Omega(P^3 \log P)$, and similarly for S, the* JOIN *algorithm is load balanced and has one MP step.*

PROOF. We will analyze each of the three cases of the Algorithm 5 separately. For any value a, let[7] $N(a) = f_R(a) + f_S(a) + f_Q(a)$ be the number of tuples from R, S and Q that

[7]We abbreviate $f_{R,x}$ with f_R, etc.

Algorithm 5: JOIN$(R(x, y), S(y, z))$

```
/* Broadcast Phase: compute RF, SF */
/* Communication Phase*/
/* CASE 1: R hashed, S duplicated */
 HR(@h1(x,y),x,y) :- R(@s,x,y), RF(x)
 DS(@*,x,z)       :- S(@s,x,z), RF(x)

/* CASE 2: S hashed, R duplicated */
 HS(@h2(x,z),x,z):- S(@s,x,z), SF(x), not RF(x)
 DR(@*,x,y)      :- R(@s,x,y), SF(x), not RF(x)

/* CASE 3: both R, S hashed */
 TR(@h3(x),x,y):- R(@s,x,y), not SF(x), not RF(x)
 TS(@h3(x),x,z):- S(@s,x,z), not SF(x), not RF(x)

/* Computation Phase */
Q(@s,x,y,z)  :- HR(@s,x,y), DS(@s,x,z)
Q(@s,x,y,z)  :- DR(@s,x,y), HS(@s,x,z)
Q(@s,x,y,z)  :- TR(@s,x,y), TS(@s,x,z)
```

contain the element a. Moreover, let $N_s(a)$ the number of these tuples sent to server s. We extend these notations for a set of elements V, that is, $N_s(V) = \sum_{a \in V} N_s(a)$.

First, consider a frequent value $a \in RF$. Then, $N(a) = f_R(a) \cdot f_S(a) + f_R(a) + f_S(a)$. Since the hashing we use is uniform, using the balls in bins argument, the expected maximum number of tuples from R containing a in any server is bounded by $2f_R(a)/P$ (as long as $f_R(a) \geq P \log P$, which holds when $r = \Omega(P^3 \log^3 P)$). In the case that $f_S(a) = 0$, we have $N(a) = f_R(a)$ and thus $\mathbb{E}[\max_s N_s(a)] \leq 2f_R(a)/P$. Otherwise, $f_S(a) \geq 1$ and

$$\mathbb{E}[\max_s N_s(a)] \leq f_S(a) + 2 \cdot \frac{f_R(a)}{P}(f_S(a)+1) \leq 8 \cdot \frac{f_R(a) f_S(a)}{P}$$

The last inequality holds since $1 \leq f_S(a)$ and $f_R(a)/P > 1$. By combining the two cases for $f_S(a)$, we have that for some constant c:

$$\mathbb{E}[\max_s N_s(a)] \leq \frac{c}{P} \cdot [f_R(a) f_S(a) + f_R(a)]$$

Similarly for a frequent value $a \in SF \setminus RF$, it holds that

$$\mathbb{E}[\max_s N_s(a)] \leq \frac{c}{P} \cdot [f_R(a) f_S(a) + f_S(a)]$$

Last, we consider the case of the non-frequent values $Y_f = \{a : \neg RF(a), \neg SF(a)\}$. We can bound the load $N(a)$ of a value $a \in Y_f$ by observing that $N(a) \leq \tau_R + \tau_S + \tau_R \cdot \tau_S \leq 4\tau_R \cdot \tau_S = 4 \cdot \sqrt{r \cdot s}/(P \log P) \leq 2 \cdot (r + s)/(P \log P)$. In this case, all tuples which contain a are sent to the same server. Thus, we can associate each value a with a ball of size $N(a)$ which is thrown u.a.r. to a server. We showed that the maximum size of such a ball is $w_{max} = 2 \cdot (r + s)/(P \log P)$. Let W be the total size of the balls, i.e. $W = N(Y_f) \leq |Q|$.

Now, we apply the lemma from [20]: the expected maximum load when we throw balls with maximum size w_{max} and total size W into P bins is maximized when we consider $B = W/w_{max}$ balls of size w_{max}. If $B \leq P \log P$, then the expected maximum number of balls on a server will be $\Theta(\log P)$; then, $\mathbb{E}[\max_s N_s(Y_f)] = \Theta(\log P) \cdot w_{max} = \Theta((r + s)/P)$. In the case that $B \geq P \log P$, it follows from the balls in bins that $\mathbb{E}[\max_s N_s(Y_f)] \leq \left(\frac{2}{P} \cdot \frac{W}{w_{max}}\right) \cdot w_{max} = \frac{2W}{P}$.

Finally, we sum for all the cases and all values a.

$$\mathbb{E}[\max_s n_s] \leq \sum_{a \in RF} \mathbb{E}[\max_s N_s(a)] + \sum_{a \in SF \setminus RF} \mathbb{E}[\max_s N_s(a)] + \mathbb{E}[\max_s N_s(Y_f)]$$

By substituting the bounds we have computed and summing, we conclude that the load is indeed balanced among the servers, that is, $\mathbb{E}[\max_s n_s] = O\left(\frac{|R|+|S|+|Q|}{P}\right)$. □

Generalizing to k-way joins for $k > 2$ is non-trivial. We illustrate the algorithm for $k = 3$, on the query $Q(x, y, z, w) = R(x, y), S(x, z), T(x, w)$. To see the difficulty, recall that, for a single join, $R(x, y), S(x, z)$, if $x \in RF$ then all tuples $S(x, z)$ are replicated to all servers: the cost of replication is justified by the size of the answer. But in the three-way join, they may not be in the answer, namely when x does not occur in $T(x, w)$. Thus, we need a second round of broadcast to compute the intersection of RF, SF, TF with the x-values occurring in R, S, T. We sketch the main parts of the algorithm.

Set the frequency threshold for R to $\tau_R = \sqrt[3]{r/P \log P}$; similarly for S, T. The broadcast phase has two rounds:

Broadcast 1: Compute and broadcast the sets RF, SF, TF, which contain all frequent elements in R, S, T respectively (using any of the algorithms in Subsection 4.1).

Broadcast 2: Compute and broadcast the intersections. We show this only for R (it works similarly for S, T):

```
RH^S(@s, x)  :-  RF(@s, x), S(@s,x,y)
RH^T(@s, x)  :-  RF(@s, x), T(@s,x,z)
RG^S(@*, x)  :-  RF^S(@s,x)   /* Broadcast */
RG^T(@*, x)  :-  RF^T(@s,x)   /* Broadcast */
RF'(@s,x)    :-  RG^S(@s, x), RG^T(@s, x)
```

Informally, each server s computes $RH_s^S = RF \cap S_s$ and $RF_s^T = RF \cap T_s$; it then broadcasts these values. Last, each server computes the final set $RF' = \left(\bigcup_s RF_s^S\right) \cap \left(\bigcup_s RF_s^T\right)$.

The communication phase is a straightforward generalization of Algorithm 5. There are four cases: (1) $x \in RF'$, (2) $x \in SF' \setminus RF'$, (3) $x \in TF' \setminus (RF' \cup SF')$ and (4) $x \notin (RF' \cup SF' \cup TF')$. We give the formal description of only the first case.

```
/* Case 1: R hashed, S, T replicated
 HR(@h(x,y),x,y) :- R(@s,x,y), RF'(x)
 HS(@*,x,z)      :- S(@s,x,z), RF'(x)
 HT(@*,x,w)      :- T(@s,x,w), RF'(x)
 ...
 Q(@s,x,y,z,w)   :- HR(@s,x,y), HS(x,z), HT(x,w)
```

Proposition 4.2. *The algorithm computing the 3-Join is load balanced and runs in one MP step.*

We give the proof in the appendix, using essentially the same techniques as in the case of the single join. This algorithm generalizes straightforwardly to arbitrary flat queries: note that the algorithm continues to use only two rounds in the broadcast phase.

We end this subsection by proving that two rounds are necessary in the broadcast phase. The proof is included in the appendix.

Proposition 4.3. *Any MP algorithm that computes the query $Q'(x, y, z) : -R(x, y), S(x, z), T(x)$ using at most one broadcast round is skewed.*

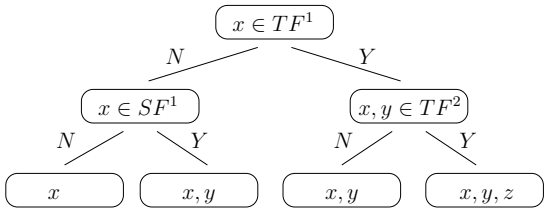

Figure 2: The decision tree for the tall query Q.

4.3 Tall Queries

We describe here the algorithm for the tall query $Q(x, y, z) : -R(x), S(x, y), T(x, y, z)$. The generalization to arbitrary tall queries is straightforward and omitted.

For the broadcast phase, we first set the thresholds $\tau_S = s/P \log P$ and $\tau_T = t/P \log P$. Using any algorithm in Subsection 4.1, we compute the following sets:

- $SF^1 \supseteq \{x \mid f_S(x) \geq \tau_S\}$
- $TF^1 \supseteq \{x \mid f_T(x) \geq \tau_T\}$
- $TF^2 \supseteq \{x, y \mid f_T(x, y) \geq \tau_T\}$

Next, each server constructs the decision tree of Figure 2. Consider a server s and any tuple t belonging to one of its fragments R_s, S_s, T_s. Depending on which of the relations t belongs to, a subset of the variables x, y, z is bound. That is, if $t \in R$ then it binds x; if $t \in S$ then it binds x, y; and if $t \in T$ then it binds x, y, z. Starting from the root, we follow the decision tree until one of two things happens:

- We reach a node asking for a variable not bound by t, e.g. if $t \in R$ and we reach a node asking for x, y. In this case, we broadcast t to all the servers.
- We reach a leaf of the tree. Then, we hash t according to the hash function with parameters the variables of the leaf node, that is, we send t either to $h1(x)$ or to $h2(x, y)$ or to $h3(x, y, z)$.

For example, consider a tuple $S(a, b)$, such that a is frequent in T and a, b is not frequent in T. Then, this tuple will be sent to server $h2(a, b)$. After distributing the tuples, each server locally computes the join.

Theorem 4.4. *For $|S|, |T|$ large enough: $|S|/\log|S| = \Omega(P^2 \log P)$ (similarly for T), the algorithm for computing the tall query Q is load balanced and has one MP step.*

Proof. First, consider all the tuples that are broadcast by the algorithm. R may broadcast at most $P \log P$ tuples for frequent values in S and T. Moreover, S may broadcast a tuple for a frequent value in T, which are at most $P \log P$. Hence, each server receives at most $3 \cdot P \log P = O(n/P)$ such tuples, since $n = \Omega(P^2 \log P)$.

Thus, it suffices to measure the load caused by the tuples which are hashed. We partition the tuples into equivalence classes (*balls*), according to the values they are hashed on (notice that it suffices to consider only input tuples, since the output size is at most the input size). For example, ball $B(a)$ contains all tuples from R, S, T which are hashed only to $h1(a)$. Notice also that if $a \in TF^1$, then $B(a)$ is empty. The following properties hold for any ball:

1. Every tuple of a ball is sent to the same server.
2. Every ball is sent to a u.a.r. chosen server.
3. The maximum size w_{max} of a ball is $(s+t)/(P \log P)$.
4. The total size of the balls is $W \leq |R| + |S| + |T|$.

We only prove property (3), since the others are straightforward. Consider the case for a ball $B(a)$. Any tuple from S belonging to $B(a)$ is hashed only on the x variable; hence, by the structure of the decision tree, $a \in SF^1$, which implies that $f_S(a) < \tau_S$; hence, there exist at most τ_S such tuples. Similarly, we have at most τ_T tuples from T in $B(a)$. Thus, $|B(a)| < 1 + \tau_S + \tau_T$. For a $B(a,b)$ ball, we have that $|B(a,b)| < 2 + \tau_T$ (and for $B(a,b,c)$ the size is constant).

Following from these properties, the expected maximum load is maximized in the case of W/w_{max} balls with size w_{max}. Using the same argument as in the proof of Theorem 4.1, we obtain that $\mathbb{E}[\max_s n_s] = \Theta(\frac{|R|+|S|+|T|}{P})$. □

5. THE MAIN ALGORITHM

In this section, we show how the techniques presented in the previous sections can be combined to build a load balanced algorithm for any tall-flat query.

For ease of exposition, let us denote by $x_{1:k}$ the sequence of variables or values $x_1, x_2, \ldots, x_k$. We will also assume w.l.o.g. the following form for tall-flat queries: $Q(x_{1:k}, y_{1:\ell}) = R_1(x_1), \ldots, R_k(x_{1:k}), S_1(x_{1:k}, y_1), \ldots, S_\ell(x_{1:k}, y_\ell)$. For simplicity, we will also refer to $S_1, \ldots, S_\ell$ as $R_{k+1}, \ldots, R_{k+\ell}$.

For the algorithm, we will assume that R_i, S_i are large enough: in particular, we will need that $|R_i|/\log|R_i| = \Omega(P^2 \log P)$ and that $|S_i|/\log^\ell |S_i| = \Omega(P^{l+1} \log P)$. We also define the frequency thresholds $\tau_{R_i} = |R_i|/P \log P$ and $\tau_{S_i} = \sqrt[\ell]{|S_i|/P \log P}$.

In order to distribute the tuples in a load balanced way, the algorithm considers two cases: (1) values which cause a large load to the query result Q and (2) the rest of the values. Intuitively, we use flat-query techniques to deal with the first case and tall-query techniques for the second case.

Broadcast Phase

Broadcast 1: Using any of the algorithms in Subsection 4.1, compute the sets

$$RF_i^j \supseteq \{x_{1:j} \mid f_{R_i}(x_{1:j}) \geq \tau_{R_i}\} \quad (i = 1,k;\ j = 1,i)$$
$$SF_i^j \supseteq \{x_{1:j} \mid f_{S_i}(x_{1:j}) \geq \tau_{S_i}\} \quad (i = 1,\ell;\ j = 1,k)$$

In particular, using *Frequency Sampling*, we can compute these sets by sampling only once. One can easily check that the communication is $O(n^\varepsilon)$.

Broadcast 2: For every relation $V \neq S_i$, server s computes $SF_i(V,s) = SF_i^k \cap V_s$, i.e. intersects SF_i^k with its local copy of V. These sets are then broadcast. Finally, each server computes the sets $SF_i = \bigcap_{V \neq S_i} \left(\bigcup_s SF_i(V,s)\right)$.

Communication Phase

1. For $i = 1, \ldots, \ell$: for every $x_{1:k} \in SF_i \setminus \bigcup_{j<i} SF_j$, send every tuple of S_i containing $x_{1:k}$ to the server $h(x_{1:k}, y_i)$. Broadcast every tuple $t \in S_j, j \neq i$ containing $x_{1:k}$.
2. Consider a tuple $x_{1:q}, q \leq k$, which does not belong to case (1). For any S_i, all tuples with the same value $x_{1:k}$ are hashed to the same server. In order to decide where to hash (or broadcast) the tuple, we construct a decision tree, which generalizes the one in Figure 2. Initially, let $i = k+\ell$ and $j = 1$ (*root*). At each step, we check whether $x_{1:j} \in RF_i^j$: if this holds, we increment j (*right child*), else we decrement i (*left child*). The algorithm stops when either $j > q$, in which case we broadcast $x_{1:q}$, or when $i = 0$ (a *leaf* is reached) and the tuple is hashed to $h(x_{1:j})$.

Computation Phase

The query is locally computed at each server.

Theorem 5.1. *The algorithm computes any tall-flat query in a load balanced way and has one MP step.*

Proof. We first analyze the load for the tuples that fall into the first case of the communication phase. Let us consider the case of the FOR loop where $i = 1$ (the same argument can be applied for every i). Consider a value $x_{1:k} \in SF_1$. The total load attributed to this value is $N(x_{1:k}) = \prod_{i=1}^{\ell} f_{S_i}(x_{1:k}) + \sum_{i=1,\ell} f_{S_i}(x_{1:k})$. Since any tuple from S_1 containing $x_{1:k}$ is hashed on $(x_{1:k}, y_1)$, using the balls in bins argument, we have $\mathbb{E}[\max_s N_s(x_{1:k})] \leq \sum_{i>1} f_{S_i}(x_{1:k}) + (2 f_{S_1}(x_{1:k})/P) \cdot (1 + \prod_{i=2}^{\ell} f_{S_i}(x_{1:k}))$. Moreover, notice that $f_{S_i}(x_{1:k}) \geq 1$ is guaranteed from the second step of the broadcast phase and also $f_{S_1}(x_{1:k}) \geq P$. Thus,

$$\mathbb{E}[\max_s N_s(x_{1:k})] \leq \frac{2k}{P} \cdot \prod_{i=1}^{\ell} f_{S_i}(x_{1:k})$$

We next consider the values of the second case. In order to compute the load from the tuples that are broadcast to all servers, notice that we have $O(P \log P)$ frequent values for each $x_{1:j}$ in relation R_i, hence a total load of $O(k^2 P \log P)$ for each server. As for relations S_i, we have $|S_i|/\tau_{S_i}$ frequent values for each $x_{1:j}$ in S_i. Thus, each relation causes a load of $O(k \cdot |S_i|/\tau_{S_i})$ to each server from broadcasting tuples. As long as $|S_i| \geq P^{\ell+1} \log P$, this load is bounded by $O(|S_i|/P)$.

Finally, let us also measure the load caused by the tuples which are hashed. We partition the tuples into equivalence classes (*balls*), according to the values they are hashed with. Notice that every ball has the following properties: (1) every tuple of a ball is sent to exactly one server, and (2) every ball is sent to a u.a.r. chosen server.

Now, consider a ball $B(x_{1:q}), q \leq k$. We define the size of the ball to include the size of the output. Consider any tuple t from a relation R_i belonging to $B(x_{1:q})$. Clearly, for $i \leq q$, each relation R_i sends at most one tuple to this ball. For $R_i, i > q$, by the structure of the decision tree, $f_{R_i}(x_{1:q}) < \tau_{R_i}$ and thus there exist at most τ_{R_i} such tuples. The same holds for any S_i. Thus,

$$|B(x_{1:q})| \leq q + \sum_{i=q+1}^{k} \tau_{R_i} + \sum_{i=1}^{\ell} \tau_{S_i} + |Q_{B(x_{1:q})}|$$

where $Q_{B(x_1)}$ denotes the tuples of Q produced by the tuples in $B(x_{1:q})$. Next, notice that

$$|Q_{B(x_{1:q})}| = \sum_{x_{q+1:k} \in B(x_{1:q})} \prod_{i=1}^{\ell} f_{S_i}(x_{1:q}, x_{q+1:k})$$

Since every tuple is hashed only on $x_{1:q}$, it must hold that $\sum_{x_{q+1:k} \in B(x_{1:q})} f_{S_i}(x_{1:q}, x_{q+1:k}) \leq \tau_{S_i}$. This implies that $|Q_{B(x_{1:q})}|$ is maximized when there exists a value $x'_{q+1:k}$ such that for every S_i, $f_{S_i}(x_{1:q}, x'_{q+1:k}) = \tau_{S_i}$ (and zero for all the other values). Then, using the Arithmetic-Geometric means inequality, we obtain that

$$|Q_{B(x_{1:q})}| \leq \prod_{i=1}^{\ell} \tau_{S_i} = \frac{\prod_{i=1}^{\ell} s_i^{1/\ell}}{P \log P} \leq \frac{1}{\ell} \cdot \frac{\sum_{i=1}^{\ell} s_i}{P \log P}$$

This implies that $|B(x_{1:q})| \leq c \cdot \frac{size(D)}{P \log P}$, ($c$ is some constant); hence $w_{max} = c \cdot \frac{size(D)}{P \log P}$.

Let W be the total load of the balls. The expected maximum load is maximized when we have just W/w_{max} balls with size w_{max}. Using the same argument as in the proof of Theorem 4.1, we obtain that the expected maximum weight attributed to the hashed values is $\Theta(W/P)$. Summing over all cases, we conclude that the algorithm is indeed load balanced. □

6. IMPOSSIBILITY RESULTS

In this section we prove that any query that is not tall-flat cannot be computed in one MP step. First, we show this result for two specific queries, $RST(x,y) : -R(x), S(x,y), T(y)$ and $J(x,y) : -R(x), S(x), T(y)$. Using these results, we prove the claim for any query that is not tall-flat.

THEOREM 6.1. *Any algorithm that computes the query $RST(x,y) : -R(x), S(x,y), T(y)$ in one MP step is skewed.*

PROOF. Let A be a one step, load balanced algorithm for RST. Recall from the proof of Proposition 3.5 that we assume that R, S, T are partitioned separately. Using Corollary 3.8, we construct vectors $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_n)$, such that we can substitute each value x_i (y_i) with a value x_i' (y_i') and obtain a colluding set. Now, let us consider an instance D of the database where $R = X$, $T = Y$ and $S = \{(x_i, y_i) \mid i = 1, \ldots, n\}$. Fix also a run $h \in \mathcal{H}$ such that the computation for this instance is load balanced.

Let us first consider the broadcast phase. Since R, T contain disjoint elements and the information exchanged may be only generic (i.e. obtained by generic computations on the instance), the only way for the servers to gain any information about the other relations is to exchange tuples $R(x_i)$, or $T(y_i)$, or $S(x_i, y_i)$: we say that the values $x_i \in X$ or $y_i \in Y$ or both have been *broadcast*. The total amount of communication is bounded by $O(n^\varepsilon)$, hence at most $O(n^\varepsilon)$ values from X and from Y are broadcast. Let X' and Y' be the other values, that are not broadcast: hence $|X'| = n - O(n^\varepsilon)$ and similarly for $|Y'|$. Let $S' = \{(x_i, y_i) \mid x_i \in X', y_i \in Y'\}$; it follows that $|S'| = n - O(n^\varepsilon)$.

Claim: Suppose that we replace S by S_π such that $S_\pi = \{(x_i, y_i) \mid (x_i, y_i) \notin S'\} \cup \{(x_i, y_{\pi(i)}) \mid (x_i, y_i) \in S'\}$, where π is any permutation on the indices of S'. Let us call this instance D_π. Using the argument in Corollary 3.8, we can prove that every server containing R, T tuples will receive exactly the same information under D or D_π.

Thus, the tuples of R, T will be distributed as before, that is, in a load balanced way. Denote $R' = X'$ and $T' = Y'$.

We say that a pair $(R'(x_i), T'(y_j))$ meets when both tuples are placed in the same server. We next compute the total number of pairs which meet for an instance D_π. Since each server holds $O(n/P)$ tuples from R', T', it contains at most $O(n^2/P^2)$ pairs that meet, which gives us a total of $O(n^2/P)$ pairs. However, drawing values from R', T', there are $O(n^2)$ possible pairs. By the pigeonhole principle, there exists a pair such that $R(x_i), T(y_j)$ are not placed in the same server. Then, we fix a permutation π such that the tuple $S(x_i, y_j)$ appears in S_π.

Finally, let us examine the instance $D_\pi : R, T, S_\pi$. For this instance, the tuple (x_i, y_j) is included in the final result. However, the server s where $S(x_i, y_j)$ is sent can not contain both $R(x_i), T(y_j)$. Let us assume w.l.o.g. that it does not contain $R(x_i)$. Then, consider the instance D_π' where we substitute the tuple $R(x_i)$ with $R(x_i')$.

Following from the collusion property, and since x_i' or x_i are not communicated during the broadcast phase, the computation will be identical. This time, however, the tuple (x_i, y_j) does not belong in the output. Nevertheless, the server s that decides upon whether to output the tuple or not does not contain x_i'; due to the genericity of the computation, s must again output the tuple (x_i, y_j), which leads to a contradiction. □

THEOREM 6.2. *Any algorithm that computes the query $J(x,y) : -R(x), S(x), T(y)$ in one MP step is skewed.*

PROOF. Let A be an algorithm computing J. First, we fix the instance of $T = \{y_1, \ldots, y_n\}$. Applying Corollary 3.8, we can then find a vector $X = (x_1, \ldots, x_n)$ with distinct elements, such that for each element $x_i \in X$, there exists a vector X_i' where x_i is replaced by $x_i' \neq x_i$ and X, X_i' collude.

Now, let us consider an instance D of the database where $R = X$ and $S = \{x_i' \mid x_i \in X\}$. Following our construction, R and S are disjoint. This means that J is empty and thus the total load is $\sum_s n_s = O(n)$. Since A is load balanced, there exists a run $h \in H$ such that the computation is load balanced.

Notice that, since the elements are disjoint within each relation and we allow only generic computation during the broadcast phase, the only way to gain information is by sending tuples. However, the amount of tuples we can send is limited to $O(n^\varepsilon)$. This means that from each relation, at most $O(n^\varepsilon)$ tuples can be sent to other servers. From this point, we consider only the tuples $t \in R$, such that t, t' are not communicated (call these set R'). Clearly, $|R'| = n - O(n^\varepsilon)$. We similarly define S'.

We say that a pair of elements (x_i, y_j) meets when $T(y_j)$ is placed in the same server with either $R'(x_i)$ or $S'(x_i')$. Since each server receives at most $O(n/P)$ values from each relation, each server holds $O(n^2/P^2)$ pairs that meet; in total, $O(n^2/P)$ pairs. However, the total number of possible pairs is $O(n^2)$. This implies that, by the pigeonhole principle, we can find a pair (x_i, y_j) such that the tuple $T(y_j)$ does not occur in the same server with either $R(x_i)$ or $S(x_i')$.

For the last step of the proof, consider an instance D' where we replace in R the value x_i with x_i'. The servers will receive exactly the same information, since x_i or x_i' are not communicated during the broadcast phase. Thus, due to the collusion property, the behavior of the algorithm will be identical and, for the new instance, $T(y_j)$ will not be placed in the same server as $R(x_i'), S(x_i')$. Consequently, A will not include the tuple (x_i', y_j) in the query output, which is a contradiction, sine (x_i', y_j) now belongs in the final result. □

Based on Theorem 6.1 and Theorem 6.2, we can now prove the following characterization.

THEOREM 6.3. *Let Q be any query that is not tall-flat. Any algorithm that computes Q in one MP step is skewed.*

PROOF. We first show that the queries computable in one MP step are a subset of hierarchical queries. Indeed, consider a query Q which is not hierarchical. Then, there exists a pair of variables x, y such that $at(x) \cap at(y) \neq \emptyset$ and none of $at(x), at(y)$ is a subset of the other. This means that w.l.o.g. we can find atoms R, S, T such that $at(x) \setminus at(y) = \{R\}$, $at(y) \setminus at(x) = \{T\}$ and $at(x) \cap at(y) = \{S\}$. Fix any variable $\neq x, y$ to obtain the same constant value in any relation. Then, Q reduces to computing the query $RST(x,y) : -R(x), S(x,y), T(y)$, which by Theorem 6.1 cannot be computed in one MP step.

We also have the following property: if $|at(x)| > 1$ for a variable x, then for every variable y we have that $at(x) \cap at(y) \neq \emptyset$. Indeed, if that was not true, then by fixing all other variables $\neq x, y$ to obtain the same constant value, we reduce Q to $J(x,y) : -R(x), S(x), T(y)$, which cannot be computed in one MP step by Theorem 6.2. Thus, if x appears in more than one relation, then for every variable $y \neq x$ it holds that either $at(x) \subseteq at(y)$ or $at(y) \subseteq at(x)$.

Let us now order the m variables in decreasing order of $|at(x)|$: $x_1, x_2, \ldots, x_m$. Consider the smallest index k such that for every $i > k$, $|at(x_i)| = 1$. Using the above property,

for the first k variables we have that $at(x_1) \supseteq at(x_2) \ldots \supseteq at(x_k)$. Finally, consider a variable $x_j, j > k$; by construction $|at(x_j)| = 1$. We use again the property to see that for any $i \leq k$, $at(x_j) \subseteq at(x_i)$, since $at(x_j)$ is a singleton set. Thus, Q satisfies all three properties of a tall-flat query. □

7. DISCUSSION

Star queries. Consider the following question: given a query Q, what is the smallest number of MP steps necessary to compute Q ? We answer this question for *star queries*, and leave it open for arbitrary conjunctive queries. W.l.o.g. we assume that the star query is of the form $Q(x_1, \ldots, x_k) : -R(x_1, \ldots, x_k), S_1(x_1), \ldots, S_k(x_k)$. Since Q is not tall-flat, we need at least two MP steps to compute it. We show here that two steps are indeed optimal.

PROPOSITION 7.1. *A star query can be computed in two MP steps.*

PROOF. The load balanced algorithm for any star query works as follows. In the first step, compute all subqueies $Q_i(x_1, \ldots, x_k) : -R(x_1, \ldots, x_k), S_i(x_i)$ in parallel, in one MP step (each is a semijoin query). In the second step, compute the intersection $Q(\bar{x}) : -Q_1(\bar{x}), Q_2(\bar{x}), \ldots, Q_k(\bar{x})$ (this MP step does not need a broadcast phase). □

General queries. If Q is a general conjunctive query, it seems difficult to determine the minimum number of parallel steps required to compute Q, because the intermediate results may be much larger than the output. For example, the output may be empty, but in any query plan, some subplan may return an intermediate result whose size is quadratic in the size of the input. One possibility is to adjust the definition of load balance to include in n the size of any intermediate computations.

An algorithm for this case would work as follows. Let us assume a query Q and a query plan P for Q. First, reduce the nodes of P by combining consecutive query computations which correspond to a tall-flat query in one MP step. For example, if the plan operator computes $P_1(x) : -R(x), S(x)$ and its parent computes $P_2(x) : -P_1(x), T(x)$, then we compress the computation in one step: $P_2(x) : -R(x), S(x), T(x)$, since it is a flat query. Each plan P can be thus converted into a minimal plan P'; let d be its depth. It is easy to observe that one can compute P' in d MP steps.

EXAMPLE 7.2. *Let us consider the following chain query: $C(x, y, z, w, v) : -R(x, y), S(y, z), T(z, w), U(w, v)$. A naive query plan would sequentially compute the 3 joins to get the final result: this requires 3 MP steps. However, we can do better if we consider the following query plan:*

$$(R(x, y) \bowtie S(y, z)) \bowtie (T(z, w) \bowtie U(w, v))$$

This plan corresponds to a query tree of depth 2; notice that $R \bowtie S$ and $T \bowtie U$ can be computed in parallel in the same MP step. Hence, we can compute the query in just two MP steps, which is also optimal (since it is not tall-flat).

This example generalizes to any chain query: a chain query with k atoms can be computed in $\log k$ MP steps.

Using Data Statistics. So far, our discussion has focused on the worst case scenario, and both our algorithms and our impossibility results assumed that the sizes of the input relations are independent. In practice, one often knows the sizes of the relations, and can determine that one is much larger than the other. Afrati and Ullman [1] have shown how to compute any conjunctive query in one parallel step. We illustrate their algorithm on the k-way star query Q at the beginning of this section. Assume that the P processors are organized in a k-dimensional grid. Thus, each server is indexed by a k-tuple of integers $(s_1, \ldots, s_k)$, where $s_i \in [\sqrt[k]{P}]$, for $i = 1, k$. For each $i = 1, k$, let $h_i : U \rightarrow [\sqrt[k]{P}]$ be a hash function. Then, during the communication phase the algorithm sends $R(x_1, \ldots, x_k)$ to the server $(h_1(x_1), \ldots, h_k(x_k))$ and replicates $S_i(x_i)$ to all servers of the form $(*, *, \ldots, h_i(x_i), \ldots, *)$; in other words, $S_i(x_i)$ is replicated $\sqrt[k]{P^{k-1}}$ times. In the computation phase, each server computes the join locally. Thus, the algorithm takes a single communication step. In our framework, the algorithm is skewed, because it replicates the relations S_i. However, suppose that it holds $|R| = n$, $|S_1| = \ldots = |S_k| = m$, and $m\sqrt[k]{P^{k-1}} = O(n)$; then the algorithm is load balanced, and computes the star query in one step. This discussion shows that better MP algorithms can be designed by taking into account statistics on the input relations.

8. CONCLUSIONS

In this work, we propose a new theoretical model which captures massive parallelism in today's systems. In this model, the measure of complexity consists of the number of parallel steps necessary to compute a query. Under this context, we study the complexity of conjunctive queries and we give a complete characterization of the queries computable in one parallel step. Future work may include several directions: for any given conjunctive query, can we find the most efficient query plan in terms of steps of the MP model; what is the parallel complexity for more general sets of queries (e.g. queries with unions); and, finally, how can we implement and make the algorithms presented here practical.

Acknowledgments. We would like to thank Foto Afrati, Magda Balazinska, Bill Howe, YongChul Kwon and Jeffrey Ullman for the useful discussions and suggestions.

9. REFERENCES

[1] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.
[2] P. Alvaro, W. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. C. Sears. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.
[3] L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.
[4] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265–1276, August 2008.
[5] S. Cohen. Containment of aggregate queries. *SIGMOD Record*, 34(1):77–85, 2005.
[6] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *PPOPP*, pages 1–12, 1993.
[7] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544, 2007.
[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[9] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
[10] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.
[11] M. Grohe, Y. Gurevich, D. Leinders, N. Schweikardt, J. Tyszkiewicz, and J. V. den Bussche. Database query processing using finite cursor machines. *Theory Comput. Syst.*, 44(4):533–560, 2009.
[12] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[13] N. Immerman. Expressibility and parallel complexity. *SIAM J. Comput.*, 18(3):625–638, 1989.
[14] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.
[15] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. Research Report UW-CSE-11-03-01, University of Washington, 2011.
[16] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.
[17] 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.
[18] A. Pagh and R. Pagh. Uniform hashing in constant time and optimal space. *SIAM J. Comput.*, 38(1):85–96, 2008.
[19] M. Raab and A. Steger. "balls into bins" - a simple and tight analysis. In *RANDOM*, pages 159–170, 1998.
[20] P. Sanders. On the competitive analysis of randomized static load balancing. In *Proceedings of the first Workshop on Randomized Parallel Algorithms, RANDOM*, 1996.
[21] L. J. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984.
[22] 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.
[23] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
[24] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1043–1052, New York, NY, USA, 2008. ACM.
[25] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

APPENDIX

PROPOSITION A.1. FREQUENCY SAMPLING *(Subsection 4.1) includes in F every value x with $f_R(x) \geq \tau$ with high probability. Moreover, for some constant b, F contains no value x' with $f_R(x') < b \cdot \tau$ w.h.p.*

PROOF. The proof is based on a standard application of Chernoff bounds. For the first part, we compute the probability that a frequent value a is not included in F. The probability that we sample a tuple with this value is $f_R(a)/r \geq \tau/r$ and the probability that $a \notin F$ equals the probability that tuples with a are sampled less than $c\cdot\tau\cdot T/r$ times. If $K(a,v)$, we denote v by $f'(a)$, i.e. $f'(a)$ gives the *estimation* for the frequency of a. Then, $Pr[a \notin F] = Pr[f'(a) < c\cdot\tau\cdot T/r]$.

Notice that each sample is an independent random choice and a is chosen with probability $\geq \tau/r$. Moreover, $\mathbb{E}[f'(a)] = \sum_{j=1,T} f_R(a)/r \geq \tau\cdot T/r$. We can thus apply the Chernoff bound with $\delta = (c-1)/c$, which gives us that $Pr[f'(a) < \frac{c\cdot\tau\cdot T}{r}] \leq Pr[f'(a) < (1-\delta)\cdot\mathbb{E}[f'(a)]] < e^{-\mathbb{E}[f'(a)]\delta^2/2}$. Since there are $\leq r/\tau$ frequent values, we can apply a union bound: $Pr[\exists a \notin F] \leq \sum_{a:f_R(a)\geq\tau} Pr[a \notin F] < (r/\tau)e^{-\mathbb{E}[f'(x)]\delta^2/2} = (r/\tau)e^{-C\cdot\tau\cdot T/r}$, where $C = (c-1)^2/2c^2$. For $T = d_1 \cdot \frac{r\log r}{\tau}$, where d_1 is a constant, the probability of failure becomes arbitrarily small.

For the second part, consider a value a such that $f_R(a) < b\cdot\tau$ for some constant $b < 1$. Using similar arguments as above, we can show that $\mathbb{E}[f'(a)] < b\cdot\tau\cdot T/r$. We apply again the Chernoff bound (for a suitable choice of δ): $Pr[a \in F] = Pr[f'(a) \geq \frac{c\cdot\tau\cdot T}{r}] = Pr[f'(a) > (1+\delta)\cdot\mathbb{E}[f'(a)]] < 2^{-(1+\delta)\cdot\mathbb{E}[f'(a)]} = 2^{-C\cdot\tau\cdot T/r}$, where C is some constant depending on b, c. We use again a union bound to get that $Pr[\exists a \in F] \leq \sum_{a:f_R(a)<b\cdot\tau} Pr[a \in F] \leq r\cdot 2^{-\frac{C\cdot\tau\cdot T}{r}}$. For $T = d_2 \cdot \frac{r\log r}{\tau}$, where d_2 is an appropriate constant, the probability of failure becomes arbitrarily small ($O(1/r)$).

By choosing $d = \max\{d_1, d_2\}$, for $T = d\cdot\frac{r\log r}{\tau}$, both parts of the proposition are satisfied w.h.p. □

PROOF. (Of Proposition 4.3) Recall that we assume separate partitioning of the input relations. Let A be a load balanced algorithm computing Q with one broadcast round.

Moreover, let us denote by $\mathcal{H}$ the set of hash families from where A randomly picks a hash family. Applying Corollary 3.8, we can find a vector $X = (x_1, \ldots, x_n)$ such that for each $x_i \in X$, there exists a vector X_i' where x_i is substituted by a value $x_i' \neq x_i$ and X, X_i' collude.

Now, consider any instance where $T = X$. During the broadcast step, the servers containing T decide, independently of the relations R, S, to communicate $O(n^\varepsilon)$ tuples of X. Denote by X' the set containing the rest of the tuples, $|X'| = n - O(n^\varepsilon)$. Next, pick a value $x_i \in X'$ and consider the instance where $R = \{(x_i, y_1), \ldots, (x_i, y_n)\}$, $S = \{(x_i, z_1), \ldots, (x_i, z_n)\}$ and $T = \{x_1, \ldots, x_i', \ldots, x_n\}$.

Since the computation is load balanced, and the output empty, there exists a run $h \in \mathcal{H}$ such that the maximum load of any server is $O(n/P)$. We say that a pair (y_j, z_k) meets when both tuples $R(x_i, y_j), S(x_i, z_k)$ are placed in the same server. There are $O(n^2)$ possible pairs; however, each server cannot hold more than $O(n^2/P^2)$ pairs that meet. Thus, in total the servers hold $O(n^2/P)$ pairs, which implies that there exists at least one pair (y_j, z_k) such that $R(x_i, y_j), S(x_i, z_k)$ are never placed in the same server.

Finally, let us examine the case where we substitute the tuple $T(x_i')$ with $T(x_i)$. Since the two vectors collude, T, R, S will send exactly the same information during the broadcast phase. Moreover, since the replaced tuple will not be communicated, servers containing R, S will distribute their tuples in exactly the same way, hence $R(x_i, y_j), S(x_i, z_k)$ will never meet. Following from the genericity of the computation, the tuple (x_i, y_j, z_k) can not belong in the output; however, since $x_i \in T$, this is a contradiction. □

PROOF. (Of Proposition 4.2) Notice that each value of x falls into exactly one of the four cases. Let us first consider the first case and a value $a \in RF'$.

We have that $N(a) = f_R(a)f_S(a)f_T(a) + f_R(a) + f_S(a) + f_T(a)$. The second step of the broadcast phase guarantees that $f_S(a), f_T(a) \geq 1$. Hence $\mathbb{E}[\max_s N_s(a)] \leq f_S(a) + f_T(a) + \frac{2f_R(a)}{P}\cdot(f_S(a)f_T(a)+1) \leq \frac{c}{P}\cdot f_R(a)f_S(a)f_T(a)$ for some constant c. We can obtain a similar bound for values of x which fall into the cases 2,3. It remains to consider the last case, where a is not frequent in any of the relations. Let X_f be the set of these values. For any value $a \in X_f$, we have that $N(a) \leq \tau_R\tau_S\tau_T + \tau_R + \tau_S + \tau_T \leq 4\tau_R\tau_S\tau_T$. Applying the Arithmetic-Geometric means inequality, we get that $\tau_R\tau_S\tau_T = \frac{\sqrt[3]{rst}}{P\log P} \leq \frac{r+s+t}{3P\log P}$. Hence, $N(a) \leq c'\cdot\frac{r+s+t}{P\log P}$ for some constant c'.

Using the balls in bins framework, we are throwing balls u.a.r. into servers and $w_{max} = c'\cdot\frac{r+s+t}{P\log P}$. Following again the argument of the proof for Theorem 4.1, we derive that $\mathbb{E}[\max_s N_s(X_f)] = \Theta(W/P)$, where W denotes the total load attributed to the set X_f.

Summing over the values of x for all cases, we obtain that the computation is indeed load-balanced. □