Logical Aspects of Massively Parallel and Distributed Systems

Frank Neven
Hasselt University and transnational University of Limburg
frank.neven@uhasselt.be

ABSTRACT

Database research has witnessed a renewed interest for data processing in distributed and parallel settings. While distributed and parallel data management systems have been around for quite some time, it is the rise of cloud computing and the advent of Big Data that presents the community with new challenges. This paper highlights recent research concerning the logical foundations of massively parallel and distributed systems. The first part of the paper concerns massively parallel systems where computation proceeds in a number of synchronized rounds. Here, the focus is on evaluation algorithms for conjunctive queries as well as on reasoning about correctness and optimization of such algorithms. The second part of the paper addresses a distributed asynchronous setting where eventual consistency comes into play. Here, the focus is on coordination-free computation and its relationship to logical monotonicity and Datalog programs.

1. INTRODUCTION

The rise of cloud computing and the popularity of Big data lead to a renewed interest for data processing in parallel and distributed settings. The purpose of this paper is to highlight research directions addressing questions related to the logical foundations of these areas. We focus on two particular settings.

The first setting is motivated by query evaluation for very large data sets where data is simply too large to be handled on a single machine. Programs for querying such data typically consist of a number of rounds (or steps) where in each round the data is reshuffled (or repartitioned) over a number of servers followed by a computation at each server. In this model there is a synchronization step after each round. We follow the MPC model as introduced by Koutris and Suciu [40] and focus on single-round MPC programs of which Shares [9] and the HyperCube algorithm [20] are typical examples. We discuss a framework for reasoning about the correctness and optimization of such programs for evaluating conjunctive queries.

PODS'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-4191-2/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2902251.2902307

The second setting is motivated by declarative distributed computing (see, e.g., [13]) where the specific data partitioning is given and data can not be reshuffled. Communication is asynchronous and queries adopt an eventually consistent semantics. The focus is on relating coordination-free computations with logical monotonicity and Datalog programs.

Outline. In Section 2, we introduce some notation. We consider the massively parallel communication (MPC) model as our prime example of the setting for synchronous computation in Section 3. We discuss parallel-correctness for single-round algorithms within MPC in Section 4. We consider the asynchronous setting in Section 5 where we focus on coordination-free computations. We conclude in Section 6.

2. PRELIMINARIES

Even though the setup of the paper is rather informal, it still makes sense to fix some of the notation and to at least formally define the most important class of queries that we consider: the class of conjunctive queries.

Throughout the rest of the paper, we assume an infinite domain **dom** and a database scheme consisting of relation names with associated arities. A *(database) instance* I is simply a finite set of facts. For a fact f, we denote by $adom(f)$ the set of **dom**-values occurring in f. Furthermore, $adom(I) = \bigcup_{f \in I} adom(f)$. A *conjunctive query (CQ)* Q is an expression of the form

$$H(\mathbf{x}) \leftarrow R_1(\mathbf{y_1}), \ldots, R_m(\mathbf{y_m})$$

where every R_i is a relation name and every $\mathbf{y_i}$ matches the arity of R_i. We require that every variable in $\mathbf{x}$ occurs in some $\mathbf{y}_i$. We refer to the *head atom* $H(\mathbf{x})$ by $head_Q$ and to the set $\{R_1(\mathbf{y_1}), \ldots, R_m(\mathbf{y_m})\}$ by $body_Q$.

We denote by $vars(Q)$ the set of all variables occurring in Q. A *valuation* for a CQ Q is a total function $V : vars(Q) \to$ **dom**. We refer to $V(body_Q)$ as the facts *required* by V. A valuation V *satisfies* Q on instance I if all facts required by V are in I. In that case, V *derives* the fact $V(head_Q)$. The *result* of Q on instance I, denoted $Q(I)$, is defined as the set of facts that can be derived by satisfying valuations for Q on I.

3. MASSIVELY PARALLEL COMMUNICATION MODEL

The Massively Parallel Communication (MPC) model was introduced by Koutris and Suciu [40] to study the parallel complexity of conjunctive queries. The presentation of this

model here is adapted from [22]. The setting is motivated by query processing on big data that is typically performed on a shared-nothing parallel architecture where data is stored on a large number of servers interconnected by a fast network. In the MPC model, computation is performed by p servers connected by a complete network of private channels. Examples of such systems include Pig [49], Hive [53], Dremel [43], and Spark [57]. The computation proceeds in steps or rounds where each round consists of two distinct phases:

- *Communication Phase:* The servers exchange data by communicating with all other servers (sending and receiving of data).
- *Computation Phase:* Each server performs only local computation (on the data currently stored at that server).

The number of rounds then corresponds to the number of synchronization barriers that an algorithm requires. The input data is initially partitioned among the p servers and every server receives $1/p$-th of the data. There are no assumptions on the particular partitioning scheme. At the end of the execution, the output must be present in the union of the p servers. As the model focuses primarily on quantifying the amount of communication there is no a priori bound on the computational power of a server.

A relevant measure is the *load* at each server which is the amount of data received by a server during a particular round. Let m be the number of tuples in the database.[1] Then, during any point in the execution of the parallel algorithm, the load should always be a number in the interval $[m/p, m]$. Here, m/p is optimal in that the data is optimally distributed whereas a load of m means that a server has access to all of the data and can (theoretically) compute any query locally. The load is usually represented as $m/p^{1-\varepsilon}$ where $\varepsilon \in [0, 1[$. Examples of optimization goals are minimizing total load (e.g., [8]) and minimizing maximum load (e.g., [40]).

At various places below, we talk about skew. The notion refers to the presence of skewed values in the database, so-called heavy hitters, whose frequency is much higher than some predefined threshold.

To get a feeling for the model, we next present simple examples of single- and multi-round algorithms in the MPC model for evaluating specific conjunctive queries.

Example 3.1. (1) Consider the query Q_1

$$H(x, y, z) \leftarrow R(x, y), S(y, z)$$

joining two binary relations R and S over a common attribute. We consider two different strategies for computing Q_1:

(1a) Let h be a hash function mapping every domain value to one of the p servers. The following single-round algorithm then computes Q_1. In the communication phase, executed by every server on its local data, every tuple $R(a, b)$ is sent to server $h(b)$ while every tuple $S(c, d)$ is sent to server $h(c)$. In the subsequent computation phase, every server evaluates Q_1 on the received data. The output of the algorithm then is the union of the results computed at the computation phase. This strategy is called a repartition join in [23]. This approach is not resilient to skew as it is quite possible that a large part of the database is sent to one server. However, in the absence of skew, for instance, when every domain element occurs at most once in every relation, the maximum load is $O(m/p)$.

(1b) Next, we present a strategy that is resilient to skew. It is based on Ullman's drug interaction example [55] and is used explicitly in the algorithm DYM-n of [6]. For simplicity, we assume that R and S consist of m tuples. The algorithm divides R and S into $p^{1/2}$ disjoint groups of size $m/p^{1/2}$. Every combination of an R-group and an S-group can now be sent to a different server (as there are p such combinations) in the communication phase. The computation phase then consists of evaluating Q_1 on the local data. The load per server is $O(m/p^{1/2})$ independent of any skew in the database.

(2) Let Q_2 be the triangle query:

$$H(x, y, z) \leftarrow R(x, y), S(y, z), T(z, x).$$

One way to evaluate Q_2 is through a cascade of binary joins leading to a two-round algorithm. That is, first joining R and S followed by a join of T. Either the approach in (1a) or (1b) can be followed. Assuming two hash functions h and h', we only exemplify the hashing strategy of (1a). In the first round, all tuples $R(a, b)$ and $S(c, d)$ are sent to server $h(b)$ and $h(c)$, respectively. The computation phase then computes the join of R and S at each server in a relation K. In the second round, each resulting triple $K(e, f, g)$ is sent to $h'(e, g)$, while each tuple $T(i, j)$ is sent to $h'(j, i)$. Again, the computation phase then joins K and T at each server. □

We note that every MapReduce [28] program can be seen as an algorithm within the MPC model since the map phase and reducer phase readily translate to the communication and computation phase of MPC. Conceptually, a MapReduce job is a pair (μ, ρ) of functions, where μ is called the map and ρ the reduce function. The execution of a job on an input dataset then proceeds in two stages. In the first stage, called the *map stage*, each fact f from the input data set is processed by μ, generating a collection $\mu(f)$ of key-value pairs of the form $\langle k : v \rangle$. The total collection $\bigcup_f \mu(f)$ of key-value pairs generated during the map phase is then grouped on the key value resulting in a number of groups, say $\langle k_1 : V_1 \rangle, \ldots, \langle k_n : V_n \rangle$ where each V_i is set of values. Each group $\langle k_i : V_i \rangle$ is then processed by the reduce function ρ resulting again in a collection of key-value pairs per group. The total collection $\bigcup_i \rho(\langle k_i : V_i \rangle)$ is the output of the MapReduce job. Finally, a MapReduce program then is a sequence of MapReduce jobs. As MapReduce provides a higher level of abstraction, it is a relevant formalism to specify MPC algorithms.

3.1 Single-round MPC

We focus next on the one-round evaluation of multi-way joins and start off with an example.

Example 3.2. Consider again the triangle query Q_2 of Example 3.1:

$$H(x, y, z) \leftarrow R(x, y), S(y, z), T(z, x).$$

[1] In, e.g., [22], a distinction is made between the number of tuples m in the database and the number of bits M needed to represent the data. We are a bit sloppy here and stick to m.

Let α_x, α_y, and α_z be positive natural numbers such that $\alpha_x\alpha_y\alpha_z = p$. Every server can then uniquely be identified by a triple in $[1, \alpha_x] \times [1, \alpha_y] \times [1, \alpha_z]$. For $c \in \{x, y, z\}$, let h_c be a hash function mapping each domain value to a number in $[1, \alpha_c]$. The algorithm then operates as follows. In the communication phase, every fact

- $R(a, b)$ is sent to every server with coordinate $(h_x(a), h_y(b), \alpha)$ for every $\alpha \in [1, \alpha_z]$ (replicating every R-tuple α_z times);
- $S(b, c)$ is sent to every server with coordinate $(\alpha, h_y(b), h_z(c))$ for every $\alpha \in [1, \alpha_x]$ (replicating every S-tuple α_x times); and,
- $T(c, a)$ is sent to every server with coordinate $(h_x(a), \alpha, h_z(c))$ for every $\alpha \in [1, \alpha_y]$ (replicating every T-tuple α_y times).

The computation phase then consists of evaluating Q_2 on the local data at each server. The algorithm is correct as for every valuation V for Q_2, mapping variables to domain values, there is a server that contains the facts

$$\{V(R(x, y)), V(S(y, z)), V(T(z, x))\}$$

if and only if the (hypothetical) centralized database contains those facts. In this sense, the algorithm distributes the space of all valuations of Q_2 over the computing servers in an instance independent way through hashing of domain values. In the special case that $\alpha_x = \alpha_y = \alpha_z = p^{1/3}$, each tuple is replicated $p^{1/3}$ times. Assuming each relation consists of m tuples and there is no skew, each server will receive $m/p^{2/3}$ tuples for each of the relations R, S, and T. So, the maximum load per server is $O(m/p^{2/3})$.

□

The technique in Example 3.2 can be generalized to arbitrary conjunctive queries and was first introduced in the context of MapReduce by Afrati and Ullman [9] under the name of Shares. The values α_x, α_y, and α_z are called shares (hence, the name of the algorithm) and the algorithm focuses on computing optimal values for the shares minimizing the total load. The latter measure is termed the communication cost in [9] and defined as the amount of data transferred from the mappers to the reducers. Afrati et al. [7] provide a generalization of the Shares algorithm incorporating skew by distinguishing tuples that are heavy hitters. In [27] it is shown that there is a trade-off between the replication rate and the reducer size for several classes of problems.

Beame, Koutris, and Suciu [20] obtained that the strategy underlying Shares is optimal when the goal is to minimize the expected maximum load per server. More specifically, let Q be a full conjunctive query. Assuming that the sizes of all relations are equal to m and under the assumption that there is no skew, the maximum load per server is bounded by $O(m/p^{1/\tau^*})$ with high probability. Here, τ^* depends on the structure of Q and corresponds to the optimal fractional edge packing (which for Q_2 is $\tau^* = 3/2$). The algorithm is referred to as HyperCube in [20]. Additionally, the authors show that the just provided bound is tight for all one-round MPC algorithms (in the absence of skew). In the case when the relations are skewed and the heavy hitters and their frequencies are known, upper and lower bounds are provided in terms of packings of residual queries obtained by specializing the query to a heavy hitter [21]. In [39], the same authors consider a different share allocation of HyperCube to obtain a worst-case optimal algorithm for multiway join processing.

Chu, Balazinska, and Suciu [26], provide an empirical study of HyperCube (in combination with a worst-case optimal algorithm for sequential evaluation) for complex join queries, and establish, among other things, that HyperCube performs well for join queries with large intermediate results. On the other hand, HyperCube can perform badly on queries with small output.

3.2 Multi-Round MPC

Multi-round MPC has been studied in less depth. We highlight some of the results on multi-round algorithms for evaluating classes of queries.

Yannakakis algorithm [58] for acyclic conjunctive queries consists of a semi-join phase aimed at eliminating dangling tuples followed by a join phase such that the sizes of the intermediate results are never larger than the final output. Afrati et al. [6] present a generalization of this idea to the distributed setting in the algorithm GYM (Generalized Yannakakis in MapReduce). The latter algorithm takes a tree decomposition of a possibly cyclic query as input, evaluates joins of relations grouped at the same node through the Shares algorithm and executes Yannakis' algorithm on the resulting tree taking advantage of the structure of the tree to perform some joins and semi-joins in parallel. The shapes of possible tree compositions (in particular, their depth) delineate trade-offs between the number of rounds and the total amount of communication. Interestingly, the approach is resilient to skew.

Neven et al. [47] provide a formalization of MapReduce where reducers are modelled as extensions of register automata [36, 48] and obtain fragments that can express the semi-join algebra and the complete relational algebra. Employing the latter model, Tan considered multi-round algorithms for evaluating relational algebra queries that are input- as well as output-balanced [52]. Afrati and Ullman investigated ways to evaluate transitive closure and Datalog in MapReduce (see, e.g., [5, 10]).

When restricted to a single round, skewed data is provably harder to process. For instance, the maximum load for a join increases from m/p to $m/p^{1/2}$ while the load for the triangle query increases from $m/p^{2/3}$ to $m/p^{1/2}$. Beame, Koutris and Suciu [39] show that for some queries, the maximum load for skewed data can be brought down to the load of skew-free data by using multiple rounds. For example, the triangle query can be computed with load $m/p^{2/3}$ in two rounds, even if the data is skewed, while it is provably at least $m/p^{1/2}$ for one round. In contrast, the join of skewed data requires a load of $m/p^{1/2}$, no matter how many rounds one is willing to spend. Finally, Beame, Koutris, and Suciu [20] present nearly matching lower and upper bounds for multiple round algorithms computing tree-like conjunctive queries on skew-free databases of a specific form called matching databases.

4. PARALLEL-CORRECTNESS

Shares and HyperCube are a particular kind of one-round MPC algorithm where the communication phase can be seen as an initial distribution or repartitioning of the data followed by the evaluation of a single query at every server. Ameloot et al. [14] present a framework for reasoning about the correctness of such one-round MPC algorithms for the evaluation of queries but under *arbitrary* distribution poli-

cies. To target the widest possible range of repartitioning strategies, the initial distribution phase is modeled by a distribution policy that can be *any* mapping from facts to subsets of servers.

In this setting, they study two fundamental problems:

Parallel-Correctness: Given a distribution policy and a query, can we be sure that the corresponding one-round algorithm will always compute the query result correctly — no matter the actual data?

Parallel-Correctness Transfer: Given two queries Q and Q', can we infer from the fact that Q is computed correctly under the current distribution policy, that Q' is computed correctly as well?

The presentation in this section is based on [16]. The latter is a more elaborate introduction to the material contained in [14].

4.1 Parallel-Correctness

Before we can start, we need to introduce a bit of notation. A *network* $\mathcal{N}$ is a nonempty finite set of node names. A *distribution policy* $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ for a network $\mathcal{N}$ consists of a universe U and a total function $rfacts_{\boldsymbol{P}}$ that maps each node of $\mathcal{N}$ to a subset of facts from $facts\big(U\big)$.[2] Here, $facts\big(U\big)$ denotes the set of all possible facts over U. A node κ is *responsible for a fact* $\boldsymbol{f}$ (under policy $\boldsymbol{P}$) if $\boldsymbol{f} \in rfacts_{\boldsymbol{P}}(\kappa)$. For an instance I and a $\kappa \in \mathcal{N}$, let $loc\text{-}inst_{\boldsymbol{P},I}(\kappa)$ denote $I \cap rfacts_{\boldsymbol{P}}(\kappa)$, that is, the set of facts in I for which node κ is responsible. We refer to a given instance I as the *global instance* and to $loc\text{-}inst_{\boldsymbol{P},I}(\kappa)$ as the *local instance on node* κ.

Notice that every primary horizontal fragmentation (see, e.g., [50]) can be modeled as a distribution policy. Consider, for instance, a range partitioning on a relation Customer that assigns tuples to network nodes determined by a threshold on the area code.

The result $[Q, \boldsymbol{P}](I)$ of the distributed evaluation in one round of a query Q on an instance I under a distribution policy $\boldsymbol{P}$ is defined as the union of the results of Q evaluated over every local instance. Formally,

$$[Q, \boldsymbol{P}](I) \stackrel{\text{def}}{=} \bigcup_{\kappa \in \mathcal{N}} Q(loc\text{-}inst_{\boldsymbol{P},I}(\kappa)).$$

Example 4.1. Let I_e be the example database instance

$$\big\{R(a,b), R(b,a), R(b,c), S(a,a), S(c,a)\big\},$$

and Q_e be the example CQ

$$H(x_1, x_3) \leftarrow R(x_1, x_2), R(x_2, x_3), S(x_3, x_1).$$

Consider a network $\mathcal{N}_e$ consisting of two nodes $\{\kappa_1, \kappa_2\}$. Let $\boldsymbol{P}_1 = (\{a, b, c\}, rfacts_{\boldsymbol{P}})$ be the distribution policy that assigns all R-facts to both nodes κ_1 and κ_2, and every fact $S(d_1, d_2)$ to node κ_1 when $d_1 = d_2$ and to node κ_2 otherwise. Then,

$$loc\text{-}inst_{\boldsymbol{P}_1,I_e}(\kappa_1) = \big\{R(a,b), R(b,a), R(b,c), S(a,a)\big\},$$

and

$$loc\text{-}inst_{\boldsymbol{P}_1,I_e}(\kappa_2) = \big\{R(a,b), R(b,a), R(b,c), S(c,a)\big\}.$$

[2]We mention that for Hypercube distributions, the view is reversed: facts are assigned to nodes. However, both views are essentially equivalent and we will freely adopt the view that fits best for the purpose at hand.

Furthermore,

$$[Q_e, \boldsymbol{P}_1](I_e) = Q_e\big(loc\text{-}inst_{\boldsymbol{P}_1,I_e}(\kappa_1)\big) \cup Q_e\big(loc\text{-}inst_{\boldsymbol{P}_1,I_e}(\kappa_2)\big),$$

which is just $\{H(a,b)\} \cup \{H(a,c)\}$.

Next, consider the alternative distribution policy $\boldsymbol{P}_2$ that assigns all R-facts to node κ_1 and all S-facts to node κ_2, then $[Q_e, \boldsymbol{P}_2](I_e) = \emptyset$. □

Definition 4.2. A query Q is *parallel-correct on instance I under distribution policy $\boldsymbol{P}$* if $Q(I) = [Q, \boldsymbol{P}](I)$. Furthermore, Q is *parallel-correct under $\boldsymbol{P}$*, if Q is parallel-correct on all instances $I \subseteq facts\big(U\big)$.

Informally, parallel-correctness states that the naive one-round evaluation algorithm that first distributes (reshuffles) the data over the computing nodes according to $\boldsymbol{P}$ and then evaluates Q in a subsequent parallel step at every computing node, always yields the correct result, for every possible instance. Notice that, since $\boldsymbol{P}$ is defined on the granularity of a fact, the reshuffling does not depend on the current distribution of the data and can be done in parallel as well.

While Definition 4.2 is in terms of general queries, in the rest of this section, we only consider (extensions of) conjunctive queries.

We first focus on a characterization of parallel-correctness. It is easy to see that a CQ Q is parallel-correct under distribution policy $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ if, for each valuation for Q, the required facts meet at some node. That is, if the following condition holds:

$$\text{For every valuation } V \text{ for } Q \text{ over } U, \text{ there is a node } \kappa \in \mathcal{N} \text{ such that } V(body_Q) \subseteq rfacts_{\boldsymbol{P}}(\kappa). \qquad (PC_0)$$

Even though Condition (PC_0) is sufficient for parallel-correctness, it is not necessary as the following example shows.

Example 4.3. We consider the CQ Q,

$$H(x, z) \leftarrow R(x, y), R(y, z), R(x, x),$$

and the valuation $V = \{x \mapsto a, y \mapsto b, z \mapsto a\}$. Let further $\mathcal{N} = \{\kappa_1, \kappa_2\}$ and let $\boldsymbol{P}$ distribute every fact except $R(a,b)$ onto node κ_1 and every fact except $R(b,a)$ onto node κ_2. Since $R(a,b)$ and $R(b,a)$ do not meet under $\boldsymbol{P}$, valuation V witnesses the failure of Condition (PC_0) for $\boldsymbol{P}$ and Q.

However, Q is parallel-correct under $\boldsymbol{P}$. Indeed, every valuation that derives a fact $\boldsymbol{f}$ with the help of the facts $R(a,b)$ and $R(b,a)$, also requires the fact $R(a,a)$ (or $R(b,b)$). But then, $R(a,a)$ (or $R(b,b)$) alone is sufficient to derive $\boldsymbol{f}$ by mapping all variables to a (or b). Therefore, if $\boldsymbol{f} \in Q(I)$, for some instance I, then $\boldsymbol{f} \in [Q, \boldsymbol{P}](I)$ and thus Q is parallel-correct under $\boldsymbol{P}$. □

It turns out that for a semantical characterization of parallel-correctness, we need the notion of minimal valuations:

Definition 4.4. A valuation V for a CQ Q is *minimal* for Q if there does *not* exist a valuation V' for Q that derives the same head fact with a strict subset of body facts, that is, such that $V'(body_Q) \subsetneq V(body_Q)$ and $V'(head_Q) = V(head_Q)$.

Example 4.5. For a simple example of a minimal valuation and a non-minimal valuation, consider the CQ Q,

$$H(x, z) \leftarrow R(x, y), R(y, z), R(x, x).$$

Both $V_1 = \{x \mapsto a, y \mapsto b, z \mapsto a\}$ and $V_2 = \{x \mapsto a, y \mapsto a, z \mapsto a\}$ are valuations for Q. Notice that both valuations agree on the head variables of Q, but they require different sets of facts. In particular, for V_1 to be satisfying on I, instance I must contain the facts $R(a,b), R(b,a)$, and $R(a,a)$, while V_2 only requires I to contain $R(a,a)$. This observation implies that V_1 is not minimal for Q. Further, since V_2 requires only one fact for Q, valuation V_2 must be minimal for Q. □

We are now ready to state the characterization of parallel-correctness for CQs.

Proposition 4.6. *[14] A CQ Q is parallel-correct under distribution policy $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ if and only if the following holds:*

$$\begin{aligned} &\text{For every minimal valuation } V \text{ for } Q \text{ over } U, \\ &\text{there is a node } \kappa \in \mathcal{N} \text{ such that} \\ &\qquad V(body_Q) \subseteq rfacts_{\boldsymbol{P}}(\kappa). \end{aligned} \tag{PC_1}$$

The latter conditions are so fundamental when reasoning over parallel-correctness that they deserve their own terminology:

Definition 4.7. For a CQ Q and a distribution policy $\boldsymbol{P}$:

- $\boldsymbol{P}$ *saturates* Q if they fulfill Condition (PC_1); and,
- $\boldsymbol{P}$ *strongly saturates* Q if they fulfill Condition (PC_0).

We note that every Hypercube distribution for a conjunctive query Q strongly saturates Q (independent of the choices of the shares and the hash functions).

The quantifier structure in Condition (PC_1) hints at a Π_2^p upper bound for the complexity of testing parallel-correctness. Of course, the exact complexity can not be judged without having a bound on the number of nodes κ and the complexity of the test $V(body_Q) \subseteq rfacts_{\boldsymbol{P}}(\kappa)$. The largest classes of distribution policies for which an Π_2^p upper bound has been established, are gathered in the set $\mathfrak{P}_{\text{npoly}}$ that contains classes $\mathcal{P}$ of distribution policies, for which each policy comes with an algorithm $\mathcal{A}$ and a bound n on the representation size of nodes in the network, respectively, such that whether a node κ is responsible for a fact $\boldsymbol{f}$ is decided by $\mathcal{A}$ *non-deterministically* in time $\mathcal{O}(n^k)$, for some k that depends only on $\mathcal{P}$.

It turns out that the problem of testing parallel-correctness is also Π_2^p-hard, even for the simple class $\mathcal{P}_{\text{fin}}$ of distribution policies, for which all pairs $(\kappa, \boldsymbol{f})$ of a node and a fact are explicitly enumerated. Thus, in a sense, Condition (PC_1) can not be simplified.

To state the results more formally, we define the following two algorithmic problems.

PCI($\mathbf{CQ}, \mathcal{P}$)	
Input:	$Q \in \mathbf{CQ}$, $\boldsymbol{P} \in \mathcal{P}$, instance I
Question:	Is Q parallel-correct on I under $\boldsymbol{P}$?

PC($\mathbf{CQ}, \mathcal{P}$)	
Input:	$Q \in \mathbf{CQ}$, $\boldsymbol{P} \in \mathcal{P}$
Question:	Is Q parallel-correct under $\boldsymbol{P}$?

Here, **CQ** denotes the class of conjunctive queries.

Theorem 4.8. *[14] Problems* PC($\boldsymbol{CQ}, \mathcal{P}$) *and* PCI($\boldsymbol{CQ}, \mathcal{P}$) *are Π_2^p-complete, for every policy class $\mathcal{P} \in \{\mathcal{P}_{fin}\} \cup \mathfrak{P}_{npoly}$.*

We note that Proposition 4.6 continues to hold true in the presence of union and inequalities (under a suitable definition of minimal valuation for unions of CQs) leading to the same complexity bounds as stated in Theorem 4.8 [33].

When considering (unions) of conjunctive queries with negation the parallel-correctness problem becomes much more involved, since it might involve counterexample databases of exponential size. We emphasize that this exponential explosion can only occur if the arity of the relations in the database schema are not a-priori bounded by some constant. Below, $\mathbf{CQ}^\neg$ and $\mathbf{UCQ}^\neg$ denote the class of (unions) of conjunctive queries with negation.

Theorem 4.9. *[33] For every class $\mathcal{P} \in \mathfrak{P}_{npoly}$ of distribution policies,* Parallel-Correct($\boldsymbol{UCQ}^\neg, \mathcal{P}$) *and* Parallel-Correct($\boldsymbol{CQ}^\neg, \mathcal{P}$) *are coNEXPTIME-complete.*

The result and, in particular, the lower bound even holds if $\mathfrak{P}_{\text{npoly}}$ is replaced by the corresponding $\mathfrak{P}_{\text{poly}}$, where the decision algorithm for pairs $(\kappa, \boldsymbol{f})$ is deterministic and in polynomial time.

We note that, as queries in $\mathbf{CQ}^\neg$ are no longer monotone, parallel-correctness requires both parallel-soundness as well as parallel-completeness [33].

The proof of the lower bounds comes along an unexpected route and exhibits a reduction from query containment for $\mathbf{CQ}^\neg$ to parallel-correctness for $\mathbf{CQ}^\neg$. Specifically, query containment asks the question whether, given two queries Q and Q', it holds that $Q(I) \subseteq Q'(I)$ for all instances I. The latter is denoted by $Q \subseteq Q'$. It is shown in [33] that query containment for $\mathbf{CQ}^\neg$ is coNEXPTIME-complete, implying coNEXPTIME-hardness for parallel-correctness as well. The result regarding containment of $\mathbf{CQ}^\neg$ answers the observation in [44] that the Π_2^p-completeness result for query containment for $\mathbf{CQ}^\neg$ mentioned in [54] only holds for fixed database schemas (or a fixed arity bound, for that matter).

4.2 Parallel-Correctness Transfer

The one-round Shares and Hypercube algorithm require a reshuffling of the data before the evaluation of each new query. In the context of multiple query evaluation, where an optimizer tries to automatically partition the base data across multiple nodes to achieve overall optimal performance for a specific workload (see, e.g., [46, 51]), it makes sense to consider scenarios in which such reshuffling can be avoided. To this end, *parallel-correctness transfer* was introduced in [14] which states that a subsequent query Q' can always be evaluated over a distribution for which a query Q is parallel-correct.

Definition 4.10. For two queries Q and Q' over the same input schema, *parallel-correctness transfers from Q to Q'* if Q' is parallel-correct under every distribution policy for which Q is parallel-correct. In this case, we write $Q \xrightarrow{\text{pc}} Q'$.

Example 4.11. The next example is borrowed from [16]. We illustrate parallel-correctness transfer with the help of the following example queries:

$$\begin{aligned} Q_1 &: H() \leftarrow S(x), R(x,x), T(x). \\ Q_2 &: H() \leftarrow R(x,x), T(x). \\ Q_3 &: H() \leftarrow S(x), R(x,y), T(y). \\ Q_4 &: H() \leftarrow R(x,y), T(y). \end{aligned}$$

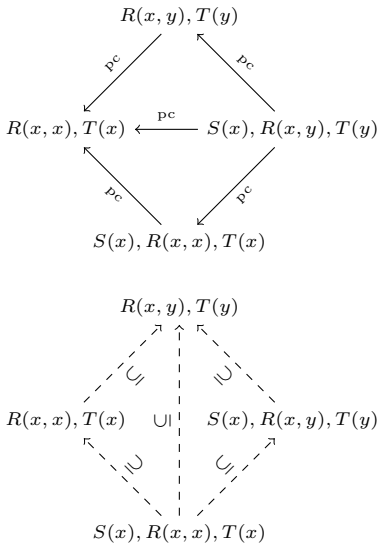

Figure 1: Relationship between the queries of Example 4.11 with respect to (a) parallel-correctness transfer and (b) query containment. (Figure taken from [16])

Figure 1 (a) shows how these queries relate with respect to parallel-correctness transfer. As an example, $Q_3 \xrightarrow{\text{pc}} Q_1$. As Figure 1 (b) illustrates, this relationships is entirely orthogonal to query containment. Indeed, there are examples where parallel-correctness transfer and query containment coincide (Q_3 vs. Q_4), where they hold in opposite directions (Q_4 vs. Q_2) and where one but not the other holds (Q_3 vs. Q_2 and Q_1 vs. Q_4, respectively). □

It turns out that, just like parallel-correctness, parallel-correctness transfer can be characterized in terms of minimal valuations. For this, we need the following notion:

Definition 4.12. For two CQs Q and Q', we say that Q *covers* Q' if the following holds:

> for every minimal valuation V' for Q', there is a minimal valuation V for Q, such that $V'(body_{Q'}) \subseteq V(body_Q)$.

Proposition 4.13. *[14] For two CQs Q and Q', parallel-correctness transfers from Q to Q' if and only if Q covers Q'.*[3]

Proposition 4.13, allows us to pinpoint the complexity of parallel-correctness transferability. For a formal statement we define the following algorithmic problem:

PC-TRANS (**CQ**)	
Input:	Queries Q and Q' from **CQ**
Question:	Does parallel-correctness transfer from Q to Q'?

When the defining condition of "covers" is spelled out by rewriting "minimal valuations" one gets a characterization with a Π_3-structure. Again, it can be shown that this is essentially optimal.

Theorem 4.14. *[14]* PC-TRANS($\boldsymbol{CQ}$) *is Π_3^p-complete.*

We note that the same complexity bounds continue to hold in the presence of inequalities and for unions of conjunctive queries [15].

It is shown in [14,15] that the complexity of deciding transferability can be lowered to NP in some cases when considering restricted classes of queries (as, for instance, the full queries) or when restricting the class of considered distribution policies (as, for instance, HyperCube distributions).

5. COORDINATION-FREE COMPUTATION

MPC is an inherent synchronized model that makes synchronization explicit by modeling computation as a sequence of rounds. In the remainder of this paper, we depart from synchronized systems and consider a setting where computing nodes are allowed to communicate freely. The particular setting we consider, is motivated by the work on declarative distributed computing, an approach where distributed computations are modeled and programmed using declarative formalisms based on extensions of Datalog (see, e.g., [1, 2, 13, 34, 35, 41, 42]).

The particular model we consider in the next section, operates under the assumption that messages can never be lost but can be arbitrarily delayed. Therefore, an *eventually consistent* semantics is adopted for queries entailing that the complete output is eventually produced after all messages have arrived and the system never outputs facts that later need to be retracted. An inherent source of inefficiency in such systems are the global barriers raised by the need for synchronization in computing the result of queries. Therefore, we consider systems that can communicate but are not allowed to coordinate, that is, they are restricted to so called coordination-free computation.

5.1 Relational Transducer Networks

In this section, we introduce relational transducer networks in a rather informal way. For a formal definition of this model we refer to [18, 19].

On a logical level, programs (queries) are specified over a global schema and are computed by multiple computing nodes over which the input database is distributed. As every node in the network runs the same program Π, we refer to a specific relational transducer network by this name. Each node in the network is assigned a part of the database, referred to as the local database. This is modeled as a horizontal distribution H mapping each node κ to a database instance such that $\bigcup_{\kappa \in \mathcal{N}} H(\kappa)$ equals the global database. In addition, every node has access to some auxiliary relations for storing data and a designated output relation. The output relation is write-only. This means that once a fact is output it can never be retracted. All data available to the node is referred to as its relational state.

Nodes can communicate asynchronously with each other via messages. The only permitted operation is a broadcast to all. However, each node has access to a relation *All* containing the names of all the nodes in the network. In this way, direct messaging can be simulated (albeit in an inefficient way) by tagging tuples with the identity of the intended receiver. As mentioned before, messages can never be lost but can be arbitrarily delayed.

[3]The terminology *saturates*, *strongly saturates*, and *covers*, was not used in [14] but was introduced in [15] and also used in [16].

Computation is modeled as a transition system. At every point in time, one node is active and can perform a transition, that is, a computation. During this computation the node reads (and removes) an arbitrary message from its input buffer, updates its relational state (possibly writing to the output relation), and sends output messages to other nodes (updating the corresponding buffers). The input message is chosen nondeterministically to model arbitrary delay of messages. Every node runs the same program Π specifying how to perform a transition (computation) based on the message read from the input buffer and the current relational state. In this paper, we assume each node has arbitrary computational power. So, Π can be any computable (but generic) function. The paper [19] also considers more restricted settings where programs Π are expressed in various logics.

A run of the system is an infinite sequence of transitions and there is a fairness criteria ensuring that no computing nodes or messages are ignored infinitely often. The output of a particular run is then the union of the facts in the output relations at every node.

It remains to specify when Π computes a particular query Q: Π computes Q iff the output of every run is $Q(I)$ for every database I, for every network and for every horizontal distribution of I. In particular, the definition implies the following:

- every run computes Q: independence of order of messages and order of execution; the infinite nature of runs together with the fact that computations are generic ensures that there is a quiescence point after which no new output is produced, hence capturing eventual consistency;
- for every network: network independence;
- for every horizontal distribution: independence of how data is initially distributed.

The following example illustrates the just introduced model and prepares for the discussion of coordination-freeness handled next.

Example 5.1. (1) Consider the query $Q_\triangle$ computing all triangles:

$$H(x,y,z) \leftarrow E(x,y), E(y,z), E(z,x), x \neq y, y \neq z, z \neq x.$$

The following program $\Pi_\triangle$ run at node κ computes Q:

1. Output all triangles in $H(\kappa)$ (recall that $H(\kappa)$ is the data local to κ).
2. Broadcast $H(\kappa)$.
3. If a new edge is received, add it to $H(\kappa)$ and output any new triangles. Repeat step (3).

It should be clear that every run of $\Pi_\triangle$ computes Q on every network and for every horizontal distribution H. The reason this naive broadcast strategy work is because $Q_\triangle$ is monotone: adding more edges to the graph can never invalidate previously output triangles. While there definitely is communication between nodes, the communication is unidirectional and does not seem to require any coordination.

(2) Now, consider the following query selecting all open triangles:

$$H(x,y,z) \leftarrow E(x,y), E(y,z), \neg E(z,x).$$

This query is not monotone. A node κ can only output an open triangle (a,b,c) when it knows that the edge $E(c,a)$ is absent on every node in the network. This means that κ somehow needs to coordinate with every other node regarding the status of $E(c,a)$. An admittedly naive but correct program replicating the previous broadcasting strategy can be constructed by adding a coordination protocol. For instance, κ could acknowledge the receipt of an edge to the sender, who in turn could inform κ when it has received the acknowledgment for all data. Nodes then start to output open triangles when they have received all data from all nodes. Notice that this requires that every node knows all other nodes participating in the network. □

It remains to define coordination-freeness. Somehow Example 5.1(2) intuitively requires coordination while 5.1(1) does not. As it makes no sense to completely prohibit communication, the notion of coordination-freeness introduced in [19] tries to separates 'data-communication' from 'coordination-communication'. More specifically,

> a relational transducer network Π computing a query Q is *coordination-free* if for every database instance I there is at least one horizontal distribution of I such that Π computes Q without ever reading input messages from input buffers.[4]

So, the definition requires the existence of some 'ideal' initial horizontal distribution on which the query can be computed without any communication. To be more accurate, nodes can send messages to other nodes, but these are never read. The intuition is then as follows: because on the ideal distribution there is no coordination needed (as even communication is not needed there), and the transducer network has to correctly compute the query on all distributions (not just on the ideal one), communication is only used to transfer data on non-ideal distributions and is not used to coordinate. Furthermore, it is useful to note that the network is not aware that the data is ideally distributed (as it can not communicate).

The program $\Pi_\triangle$ presented in Example 5.1(1) is coordination-free. The required ideal distribution is the one that assigns the complete database to every node.[5]

5.2 Monotonicity and Coordination-freeness

Hellerstein [35] and Alvaro et al. [12] formulated the CALM-principle which suggests a link between logical monotonicity on the one hand and distributed consistency without the need for coordination on the other hand. Here, CALM stands for *Consistency And Logical Monotonicity*. A crucial property of monotone programs is that derived facts must never be retracted when new data arrives. As illustrated in Example 5.1 the latter implies a simple coordination-free execution strategy: every node sends all relevant data to every other node in the network and outputs new facts from the moment they can be derived. No coordination is needed and the output of all computing nodes is consistent. This observation motivated Hellerstein [35] to formulate the CALM-conjecture which, in its revised form, states

[4]The latter is modeled by so-called heartbeat transitions in [19].

[5]We remark that in general the ideal distribution can not always give the full input to all nodes, see [19].

a query has a coordination-free execution strategy if and only if the query is monotone.[6]

In this section, we address the correspondence between coordination-freeness and logical monotonicity. More specifically, we exhibit increasingly larger classes of coordination-free computations corresponding to increasingly weaker forms of monotonicity thereby providing a fine-grained answer to the CALM-conjecture.

5.2.1 CALM-theorem

We formally define monotonicity:

Definition 5.2. A query Q is *monotone* if $Q(I) \subseteq Q(I \cup J)$ for all database instances I and J.[7]

Let $\mathcal{M}$ denote the class of all monotone queries. We denote by $\mathcal{F}_0$ the class of queries that can be computed by coordination-free relational transducer networks. Furthermore, we denote by $\mathcal{A}_0$ the class of queries that can be computed by relational transducer networks that are not aware of all the other nodes in the network, that is, have no access to the relation *All* containing the names of all the nodes in the network. We note that $\mathcal{F}_0$ is an undecidable class while $\mathcal{A}_0$ is a syntactic class. In [19], $\mathcal{A}_0$ is referred to as the class of queries computed by *oblivious* transducer networks.

Example 5.1(1) illustrates that every monotone query can be evaluated in a coordination-free way. In fact, these are also the only queries having this property as indicated by the following theorem:

Theorem 5.3. *[19]* $\mathcal{F}_0 = \mathcal{A}_0 = \mathcal{M}$.

Proof. The inclusions $\mathcal{M} \subseteq \mathcal{A}_0$ and $\mathcal{M} \subseteq \mathcal{F}_0$ follow the strategy outlined in Example 5.1. The inclusion $\mathcal{A}_0 \subseteq \mathcal{F}_0$ holds by observing that the required ideal distribution is the one that places the whole database at every node. The inclusion $\mathcal{F}_0 \subseteq \mathcal{M}$ follows as the ideal distribution determines the behavior of Π on I independent of additions to other nodes. □

5.2.2 Extensions of the CALM-theorem

The previous theorem implies that computing open triangles can not be done in a coordination-free manner. We argue that when transducer networks have more knowledge on how the initial data is distributed, the open triangle query can in effect be evaluated in a coordination-free manner. For this, we assume that the initial horizontal distribution is defined in terms of a distribution policy $\boldsymbol{P}_H$ as defined in Section 4.1 and that every computing node can query $\boldsymbol{P}_H$. More specifically, when a and b are domain values occurring in the current relational state at κ, then κ can test whether κ is responsible for $E(a,b)$, that is, $\kappa \in \boldsymbol{P}_H(E(a,b))$. We stress that κ can not query $\boldsymbol{P}_H$ for values occurring outside of the local active domain at node κ. We refer to such transducer networks as *policy-aware*. We denote by $\mathcal{F}_1$ the class of queries that can be computed by coordination-free policy-aware relational transducer networks. We can likewise define the class $\mathcal{A}_1$ as the class of queries that can be computed by policy-aware relational transducer networks without access to the relation *All*.

[6]The original conjecture replaced monotone by Datalog [19].

[7]This definition is equivalent to the more standard definition of monotonicity but serves our purpose better: a query Q is *monotone* if $Q(I) \subseteq Q(J)$ for all database instances I and J with $I \subseteq J$.

Example 5.4. The open triangle query from Example 5.1 can now be computed in a coordination-free manner as follows:

1. Broadcast $H(\kappa)$.
2. If a new edge is received, add it to $H(\kappa)$. If there are edges in $E(a,b)$ and $E(b,c)$ in $H(\kappa)$, but edge $E(c,a) \notin H(\kappa)$ and $\kappa \in \boldsymbol{P}_H(E(c,a))$ then output (a,b,c). Repeat step (2).

□

We can now generalize Theorem 5.3 to a weaker notion of monotone queries. To this end, we say that a fact f is *domain distinct* from an instance I when $adom(f) \setminus adom(I) \neq \emptyset$. That is, f should contain an element not occurring in I. Furthermore, an instance J is domain distinct from I, when every fact in J is domain distinct from I.

Definition 5.5. A query Q is *domain-distinct-monotone* if $Q(I) \subseteq Q(I \cup J)$ for all database instances I and J for which J is domain distinct from I.

We denote the class of domain-distinct-monotone queries by $\mathcal{M}_{distinct}$.[8] Notice that $\mathcal{M} \subsetneq \mathcal{M}_{distinct}$ as the next example shows.

Example 5.6. The query $Q_\triangle$ of Example 5.1 selecting all open triangles in a graph is in $\mathcal{M}_{distinct}$. Indeed, towards a contradiction assume there are instances I and J with J domain distinct from I such that $Q(I) \not\subseteq Q(I \cup J)$. This means that there is a triple $(a,b,c) \in Q(I)$ for which $(a,b,c) \notin Q(I \cup J)$ implying that $\{E(a,b), E(b,c)\} \subseteq I$ and $E(c,a) \notin I$ but $E(c,a) \in J$. However, in that case J is not domain distinct from I which leads to the desired contradiction.

Consider the query $Q_{\neg \mathrm{TC}}$ defining the complement of the transitive closure of E. We argue that $Q_{\neg \mathrm{TC}} \notin \mathcal{M}_{distinct}$. Indeed, consider the empty instance I with $adom(I) = \{a,b\}$. Then, $(a,b) \in Q_{\neg \mathrm{TC}}(I)$ but

$$(a,b) \notin Q_{\neg \mathrm{TC}}(I \cup \{E(a,c), E(c,b)\})$$

and $\{E(a,c), E(c,b)\}$ is domain distinct from I for c different from a and b. □

The following lemma says that domain-distinct-monotone queries are in fact monotone w.r.t. induced subinstances. For $C \subseteq adom(I)$, let $I_{|C} = \{f \in I \mid adom(f) \subseteq C\}$. Then:

Lemma 5.7. *For* $Q \in \mathcal{M}_{distinct}$, $Q(I_{|C}) \subseteq Q(I)$ *for all* I.

We say that a set $C \subseteq \mathbf{dom}$ is *distinct-complete* for κ, if for every fact f with $adom(f) \subseteq C$, κ received f from another node or $\kappa \in \boldsymbol{P}_H(f)$. In particular, κ knows for sure whether f belongs to the global instance I. Then the previous lemma implies that if a subset C is distinct-complete for κ, κ can safely output $Q(H(\kappa)_{|C})$. The latter leads to a simple coordination-free algorithm for every query $Q \in \mathcal{M}_{distinct}$:

1. Broadcast $H(\kappa)$.
2. If a new fact is received, add it to $H(\kappa)$. If $H(\kappa)$ contains a set C that is distinct-complete for κ, output $Q(H(\kappa)_{|C})$. Repeat step (2).

[8]Actually, $\mathcal{M}_{distinct}$ corresponds to the class of queries preserved under extensions (see, e.g., [4, 38]).

It can easily be seen that this evaluation strategy is coordination-free: an ideal distribution is the one that assigns the whole database to every node. So, while there formally is no coordination or synchronization, unlike for monotone queries the just presented strategy does entail waiting: a compute node can only produce output for distinct-complete sets.

The following theorem provides the characterization:

Theorem 5.8. *[18]* $\mathcal{F}_1 = \mathcal{A}_1 = \mathcal{M}_{distinct}$.

We can weaken monotonicity even further beyond domain-distinct-monotonicity. We say that a fact f is *domain disjoint* from an instance I when $adom(f) \cap adom(I) = \emptyset$. That is, f consists solely of elements not occurring in I. Furthermore, an instance J is domain disjoint from I, when every fact in J is domain disjoint from I.

Definition 5.9. A query Q is *domain-disjoint-monotone* if $Q(I) \subseteq Q(I \cup J)$ for all database instances I and J for which J is domain disjoint from I.

We denote the class of domain-distinct-monotone queries by $\mathcal{M}_{disjoint}$.

Example 5.10. It can easily be seen that the query $Q_{\neg \mathrm{TC}}$ from Example 5.6 is in $\mathcal{M}_{disjoint}$. Indeed, towards a contradiction assume there are instances I and J with J domain disjoint from I such that $Q(I) \not\subseteq Q(I \cup J)$. So, there has to be a pair $(a, b) \in Q(I)$ for which $(a, b) \notin Q(I \cup J)$. That is, there is no path from a to b in I while there is such a path in $I \cup J$. However, in that case J is not domain disjoint from I which leads to the desired contradiction.

Consider the query Q_{NT} returning the edge relation E when there is no three-node triangle present in the graph and the emptyset otherwise. Clearly, Q_{NT} is not in $\mathcal{M}_{disjoint}$. Indeed, set $I = \{E(a, a), E(b, b)\}$. Then $I = Q_{\mathrm{NT}}(I)$ but $Q_{\mathrm{NT}}(I \cup J) = \emptyset$ for every J domain disjoint from I containing a triangle. □

Hence, the previous example shows that

$$\mathcal{M} \subsetneq \mathcal{M}_{distinct} \subsetneq \mathcal{M}_{disjoint}.$$

The corresponding class of transducer networks is the one where the underlying distribution policies are domain-guided. More specifically, a domain assignment α for $\mathcal{N}$ is a total function from **dom** to the power set of $\mathcal{N}$. A domain assignment α now induces the domain-guided distribution policy $\boldsymbol{P}_\alpha$ where

$$P_\alpha(R(a_1, \ldots, a_k)) = \bigcup_{i=1}^{k} \alpha(a_i).$$

That is, every node in $\alpha(a)$ is responsible for every fact containing an occurrence of a. We refer to such transducer networks as *domain-guided*. We denote by $\mathcal{F}_2$ the class of queries that can be computed by coordination-free policy-aware relational transducer networks under domain-guided distribution policies. We can likewise define the class $\mathcal{A}_2$ as the class of queries that can be computed by policy-aware relational transducer networks under domain-guided distribution policies without access to the relation *All*.

There is an analog of Lemma 5.7. We say that J is a *component* of an instance I when $J \subseteq I$, $J \neq \emptyset$, $adom(J) \cap adom(I \setminus J) = \emptyset$ and J is minimal with this property. Then, the next lemma says that every domain-disjoint-monotone query is montone w.r.t. components.

Lemma 5.11. *For $Q \in \mathcal{M}_{disjoint}$, $Q(J) \subseteq Q(I)$ for all I and every component J of I.*

We say that a set $C \subseteq$ **dom** is *disjoint-complete* for κ if for every fact f with $adom(f) \cap C \neq \emptyset$, κ received f from another node or $\kappa \in \boldsymbol{P}_H(f)$. As every such C entails a union of components $H(\kappa)_{|C}$, κ can safely output $Q(H(\kappa)_{|C})$ for every domain-disjoint-monotone query Q. The coordination-free algorithm then operates at each node κ as follows (sketch):

1. Broadcast the active domain at κ, i.e., $adom(H(\kappa))$.
2. When a new domain element a is received, coordinate with at most one other node to transfer all tuples related to a. Notice that because the distribution is domain-guided, there is at least one computing node that contains all the facts related to a.
3. Evaluate Q on every disjoint-complete subset of I received at node κ.

We note that the previous algorithm is coordination-free: an ideal distribution is the one that assigns the whole database to every node. Still, it is quite obvious that the just presented algorithm does coordinate to some extend. Nevertheless, this coordination is not global but is pairwise and is determined by the concrete data distribution.

The following theorem provides the characterization:

Theorem 5.12. *[18]* $\mathcal{F}_2 = \mathcal{A}_2 = \mathcal{M}_{disjoint}$.

5.3 Datalog and Coordination-Freeness

As already remarked in Section 5.2, the original CALM-conjecture was formulated in terms of Datalog. It is therefore interesting to see how Datalog extensions correspond to classes of coordination-free queries. For a formal definition of Datalog we refer the interested reader to, for instance, [3].

Obviously, as all queries in Datalog are monotone, Theorem 5.3 implies that every Datalog query can be evaluated in a coordination-free manner. Afrati et al. [4] obtained that semi-positive Datalog, that is, Datalog extended with negation over base relations is in $\mathcal{M}_{distinct}$. Interestingly, Cabibbo [24] showed that Datalog with inequality and semi-positive Datalog extended with value invention capture the classes $\mathcal{M}$ and $\mathcal{M}_{distinct}$, respectively.

There also is a variant of Datalog that corresponds to $\mathcal{M}_{disjoint}$: semi-connected stratified Datalog. In particular, a Datalog program with negation is connected when the graph formed by the positive atoms is connected. Furthermore, a semi-connected stratified Datalog program is a stratified Datalog program where every stratum (except possibly the last one) is connected.

Example 5.13. The following program computing the complement of the transitive closure of E (query $Q_{\neg \mathrm{TC}}$ of Example 5.10) is a semi-connected stratified Datalog program:

$$\begin{array}{lcl} TC(x,y) & \leftarrow & E(x,y) \\ TC(x,y) & \leftarrow & TC(x,z), TC(z,y) \\ OUT(x,y) & \leftarrow & \mathrm{ADom}(x), \mathrm{ADom}(y), \neg TC(x,y) \end{array}$$

Here, ADom is a predicate defining the active domain of the instance.

Consider the following program defining query Q_{NT} of Example 5.10:

$$\begin{array}{lcl} T(x,y,z) & \leftarrow & E(x,y), E(y,z), E(z,x), y \neq x, y \neq z, x \neq z \\ S(x) & \leftarrow & \mathrm{ADom}(x), T(u,v,w) \\ OUT(x,y) & \leftarrow & E(x,y), \neg S(x) \end{array}$$

The above program is not semi-connected as the rule defining S is not connected. □

It is shown in [18] that when value invention is added to semi-connected stratified Datalog, the class $\mathcal{M}_{disjoint}$ is captured. Figure 2 (recycled from [18]) gives an overview of the presented results. Here, wILOG refers to Datalog extended with value invention while SP stands for semi-positive.

Finally, Ameloot et al. [17] obtained that connected stratified Datalog provides an effective syntax for Datalog programs that distribute over components. The authors also showed that under the well-founded semantics semi-connected Datalog programs with negation remain domain-disjoint-monotone and therefore in $\mathcal{F}_2$ providing a simple proof that win-move is coordination free for domain-guided transducer networks [59].

6. DISCUSSION

We considered the logical foundation of two different settings for parallel and distributed evaluation. Even within these particular settings there are quite a number of questions left unexplored and we touch upon some of these next.

Parallel-correctness and transfer.

For general conjunctive queries the complexity of parallel-correctness and transfer reside within different levels of the polynomial hierarchy. As these problems are static analysis problems, such high complexity does not rule out practical applicability. Still, it would be interesting to find tractable cases. Furthermore, even though containment and parallel-correctness transfer are incomparable, to take advantage of the large body of literature on query containment, it could be beneficial to investigate the relationship between these problems more deeply. The same holds true for the relationship with determinacy [45].

The notion of parallel-correctness is directly inspired by Hypercube where the result of the query is obtained by aggregating (through union) the evaluation of the original query over the distributed instance. Other possibilities are to consider more complex aggregator functions than union and to allow for a different query than the original one to be executed at computing nodes. Furthermore, it could be interesting to generalize the framework beyond one-round algorithms, that is, towards evaluation algorithms that comprise several rounds.

Coordination-free computation.

Figure 2 confirms that the notion of coordination-freeness as proposed by Ameloot et al [19] is a sensible one. In particular, as $\mathcal{A}_i = \mathcal{F}_i$ (for all $i = 0, 1, 2$) the notion seems to correspond to the intended semantics in that coordination-freeness avoids global synchronization barriers through the absence of knowledge about *all* the nodes in the network. That being said, the notion is not necessarily the best one as one could argue that, especially within $\mathcal{F}_1$ and $\mathcal{F}_2$, even though there is no global synchronization barrier, computing nodes are still prone to wait until complete subsets of the input data have been accumulated. Of course, this waiting is determined by the way the data is distributed and not by the evaluation algorithm.

The presented evaluation algorithms are very naive broadcast algorithms that essentially transfer all the data. It would be interesting to investigate efficient coordination-free algorithms. As an initial step, Ketsman and Neven [37] investigate more economical broadcasting strategies for full conjunctive queries without self-joins that only transmit a part of the local data necessary to evaluate the query at hand. In a sense, such approaches need to identify what part of the data is essential for answering a given query. An interesting related notion is that of scale independence, as introduced by Fan, Geerts and Libkin [31], where queries require only a relative small subset of the data whose size is determined by the structure of the query and the access methods rather than by the size of the data [25, 29, 30, 32].

Coordination-freeness is an all-or-nothing notion and might be too restrictive. It could be interesting to come up with ways to quantify coordination and relate it to Datalog programs. Interestingly, Alvaro et al. [11, 12] propose program analysis techniques to detect code fragments where coordination is perhaps overused. This way, some uses of coordination could be replaced with strategies like eventual consistency reducing the overall amount of coordination. Wang et al. [56] consider query plans incorporating failure-handling techniques for the evaluation of Datalog with bag-monotonic operators in synchronous as well as asynchronous settings.

Acknowledgments. I thank Tom Ameloot, Gaetano Geck, Bas Ketsman, Thomas Schwentick, Dan Suciu, and Tony Tan for helpful comments on a previous version of this paper.

7. REFERENCES

[1] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In *PODS*, pages 358–367, 2005.

[2] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for Web data management. In *PODS*, pages 293–304, 2011.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] F. Afrati, S. S. Cosmadakis, and M. Yannakakis. On Datalog vs polynomial time. *Journal of computer and system sciences*, 51:177–196, 1995.

[5] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *EDBT*, pages 1–8, 2011.

[6] F. N. Afrati, M. Joglekar, C. Ré, S. Salihoglu, and J. D. Ullman. GYM: A multiround distributed join algorithm. *CoRR*, abs/1410.4156v4, 2015.

[7] F. N. Afrati, N. Stasinopoulos, J. D. Ullman, and A. Vasilakopoulos. Sharesskew: An algorithm to handle skew for joins in mapreduce. *CoRR abs/1512.03921*, cs.DB, 2015.

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

[9] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1282–1298, 2011.

[10] F. N. Afrati and J. D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *EDBT*, pages 132–143, 2012.

[11] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed

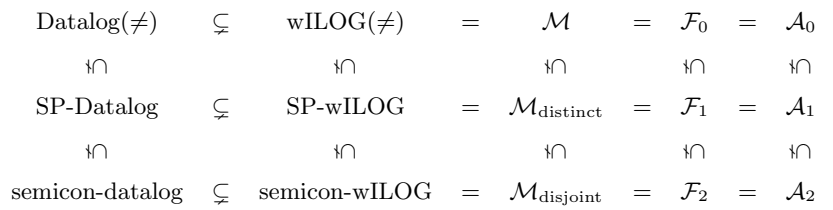

Figure 2: Correspondences between Datalog, monotone queries and classes of transducer networks.

programs. In *ICDE*, pages 52–63, 2014.

[12] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a CALM and collected approach. *CIDR*, pages 249–260, 2011.

[13] T. J. Ameloot. Declarative Networking: Recent Theoretical Work on Coordination, Correctness, and Declarative Semantics. *SIGMOD Record*, 43(2):5–16, 2014.

[14] T. J. Ameloot, G. Geck, B. Ketsman, F. Neven, and T. Schwentick. Parallel-correctness and transferability for conjunctive queries. In *PODS*, pages 47–58, 2015.

[15] T. J. Ameloot, G. Geck, B. Ketsman, F. Neven, and T. Schwentick. Parallel-correctness and transferability for conjunctive queries. Journal verion, 2015.

[16] T. J. Ameloot, G. Geck, B. Ketsman, F. Neven, and T. Schwentick. Data partitioning for single-round multi-join evaluation in massively parallel systems. *Sigmod Record*, 45(1), 2016.

[17] T. J. Ameloot, B. Ketsman, F. Neven, and D. Zinn. Datalog queries distributing over components. In *ICDT*, pages 308–323, 2015.

[18] T. J. Ameloot, B. Ketsman, F. Neven, and D. Zinn. Weaker forms of monotonicity for declarative networking: A more fine-grained answer to the CALM-conjecture. *ACM Transactions on Database Systems*, 40(4):A1–A45, 2015.

[19] T. J. Ameloot, F. Neven, and J. V. den Bussche. Relational transducers for declarative networking. *Journal of the ACM*, 60(2):15, 2013.

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

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

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

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

[24] L. Cabibbo. The expressive power of stratified logic programs with value invention. *Information and Computation*, 147(1):22–56, 1998.

[25] Y. Cao, W. Fan, T. Wo, and W. Yu. Bounded Conjunctive Queries. *PVLDB*, 7(12):1231–1242, 2014.

[26] 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.

[27] A. Das Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. In *VLDB*, pages 277–288, 2013.

[28] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[29] W. Fan and Y. Cao. An effective syntax for bounded relational queries. In *SIGMOD*, 2016.

[30] W. Fan, F. Geerts, Y. Cao, T. Deng, and P. Lu. Querying Big Data by Accessing Small Data. In *PODS*, pages 173–184, 2015.

[31] W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. In *PODS*, pages 51–62. ACM Press, 2014.

[32] W. Fan, F. Geerts, and F. Neven. Making queries tractable on big data with preprocessing. *PVLDB*, 6(9):685–696, 2013.

[33] G. Geck, B. Ketsman, F. Neven, and T. Schwentick. Parallel-correctness and containment for conjunctive queries with union and negation. In *ICDT*, pages 9:1–9:17, 2016.

[34] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In *PADL*, pages 88–103, 2010.

[35] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

[36] M. Kaminski and N. Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[37] B. Ketsman and F. Neven. Optimal broadcasting strategies for conjunctive queries over distributed data. In *ICDT*, pages 291–307, 2015.

[38] P. G. Kolaitis and M. Y. Vardi. On the expressive power of Datalog: Tools and a case study. In *PODS*, pages 61–71, 1990.

[39] P. Koutris, P. Beame, and D. Suciu. Worst-Case Optimal Algorithms for Parallel Query Processing. In *ICDT*, pages 8:1–8:18, 2016.

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

[41] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *SIGMOD*, pages 97–108, 2006.

[42] J. Ma, F. Le, D. Wood, A. Russo, and J. Lobo. A

declarative approach to distributed computing: Specification, execution and analysis. *Theory and Practice of Logic Programming*, 13(4-5):815–830, 2013.

[43] 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(1-2):330–339, 2010.

[44] M. Mugnier, G. Simonet, and M. Thomazo. On the complexity of entailment in existential conjunctive first-order logic with atomic negation. *Inf. Comput.*, 215:8–31, 2012.

[45] A. Nash, L. Segoufin, and V. Vianu. Views and queries: Determinacy and rewriting. *ACM Trans. Database Syst.*, 35(3), 2010.

[46] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, pages 1137–1148, 2011.

[47] F. Neven, N. Schweikardt, F. Servais, and T. Tan. Distributed streaming with finite memory. In *ICDT*, pages 324–341, 2015.

[48] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.

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

[50] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[51] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.

[52] T. Tan. Balancing the computation of distributed streaming with finite memory. Unpublished 2016.

[53] 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*, pages 1626–1629, 2009.

[54] J. D. Ullman. Information integration using logical views. *Theor. Comput. Sci.*, 239(2):189–210, 2000.

[55] J. D. Ullman. Designing good MapReduce algorithms. *ACM Crossroads*, 19(1):30–34, 2012.

[56] J. Wang, M. Balazinska, and D. Halperin. Asynchronous and fault-tolerant recursive Datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.

[57] 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.

[58] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.

[59] D. Zinn, T. J. Green, and B. Ludäscher. Win-move is coordination-free (sometimes). In *ICDT*, pages 99–113, 2012.