Distribution Policies for Datalog

Bas Ketsman[1]
Hasselt University, Hasselt, Belgium, and transnational University of Limburg, Belgium/The Netherlands

Aws Albarghouthi
University of Wisconsin-Madison, Madison, WI, USA

Paraschos Koutris
University of Wisconsin-Madison, Madison, WI, USA

Abstract

Modern data management systems extensively use parallelism to speed up query processing over massive volumes of data. This trend has inspired a rich line of research on how to formally reason about the parallel complexity of join computation. In this paper, we go beyond joins and study the parallel evaluation of recursive queries. We introduce a novel framework to reason about multi-round evaluation of Datalog programs, which combines implicit *predicate restriction* with *distribution policies* to allow expressing a combination of data-parallel and query-parallel evaluation strategies. Using our framework, we reason about key properties of distributed Datalog evaluation, including parallel-correctness of the evaluation strategy, disjointness of the computation effort, and bounds on the number of communication rounds.

2012 ACM Subject Classification Information systems → Query languages, Information systems → Parallel and distributed DBMSs

Keywords and phrases Datalog queries, Distributed evaluation, Distribution policies

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.17

1 Introduction

Modern data management systems – such as Spark [27, 33], Hadoop [16, 11], and others [17] – have extensively used parallelism to speed up query processing over massive volumes of data. Parallelism enables the distribution of computation into multiple machines, and thus significantly reduces the completion time for several critical data processing tasks. This trend has inspired a rich line of research on how to formally reason about the parallel complexity of join computation, one of the core tasks in massively parallel systems. Several papers [7, 8, 20, 19] have studied the tradeoff between synchronization (number of rounds) and communication cost, and have proposed and analyzed known and new parallel algorithms [4, 9]. Among these, the *Hypercube algorithm* [13, 4] can compute any multiway join query in one round by properly distributing the input data.

To reason about Hypercube-like algorithms, Ameloot et al. [6] recently introduced a framework that captures one-round evaluation of joins under different data distributions. Their framework implicitly describes a single-round parallel algorithm through a *distribution policy*, which specifies how the facts in the input relations are distributed among the machines. While for non-recursive queries a distribution policy defines a scalable parallel evaluation strategy, for Datalog programs this is typically not the case. For instance, a simple transitive

[1] PhD Fellow of the Research Foundation – Flanders (FWO)

closure query already requires for entire components of the input database that all facts must reside on the same server to ensure correctness of the computation.

To reason about Datalog evaluation in a distributed setting, we introduce a general theoretical framework that allows a combination of data and query parallelization strategies. The central concept in this framework is the notion of an *economic policy*. Our key observation is that, in order to deal with intensional predicates, we need to specify not only where a fact must be located to be *consumed* by a rule, but also where a fact must be *produced* by evaluating a rule of the program. An economic policy in our framework is defined as a pair of distribution policies: a *consumption policy*, which specifies the location of the facts that are used in the body of rules, and a *production policy*, which specifies the location of facts that appear in the head of a rule. The evaluation strategy that is implicitly defined by the data distribution must communicate any produced facts to the machines where they will be consumed, and thus can run over multiple rounds.

Our framework is inspired by a rich line of research on parallel evaluation strategies for Datalog programs from the early 90's [30, 13, 31, 34]. There, Datalog evaluation strategies are based on the idea of partitioning the instantiations of the program rules among machines by adding conditions to the bodies of the rules, called program restrictions. Some of the strategies proposed require no communication of intermediate (intensional) facts and thus can be completed in one round; other strategies require communication over multiple rounds. We show that an economic policy can capture several algorithms used for parallel evaluation of recursive and non-recursive queries, including the Hypercube algorithm [13, 4], and the decomposable strategies based on program restrictions [30].

In this framework we study several properties of economic policies. We first explore the property of *parallel-correctness*: when does an economic policy lead to a correct evaluation strategy? As can be expected, it is undecidable to show parallel-correctness for a general Datalog program, even for the simplest of economic policies. We therefore identify a sufficient condition: every minimal valuation of a rule must be *supported* by the policy. A rule valuation is supported if some machine consumes all the facts in the body, and produces the fact in the head. For unions of conjunctive queries, this condition is also necessary, recovering the result of Ameloot et al. [6]; however, we show that even for non-recursive programs with intermediate relations, the condition is no longer required. To overcome the undecidability of parallel-correctness, we identify a general family of economic policies, called *Generalized Hypercube Policies (GHPs)*, which are always parallel-correct, and further capture several commonly used parallel evaluation strategies.

Second, we study the property of *boundedness*: can we decide whether a given economic policy terminates in k rounds, independent of the input size? We show that there exists a sharp increase in complexity as we move from $k = 1$ to $k \geq 2$. For $k = 1$, we can succinctly characterize the structure of a policy that always terminates in one step. Additionally, given a GHP, we can do this in polynomial time in the description of the GHP. On the other hand, for $k \geq 2$ it is undecidable to determine whether it terminates in at most k steps, even for a GHP. We then ask which Datalog programs admit economic policies that are bounded by one round: we show that such programs are characterized by a syntactic property called *pivoting*, which was also identified by Wolfson and Silberschatz [32] in the context of decomposable programs.

2 Related Work

2.1 Parallel Complexity

The parallel complexity of Datalog was first investigated by Cosmadakis and Kanellakis [10, 18]. Later work used the complexity class NC to theoretically capture which Datalog programs are efficiently parallelizable. Since Datalog evaluation is P-complete, it is unlikely that every Datalog program belongs in NC, which implies that certain Datalog programs may not be significantly sped up through parallelism. Ullman and Van Gelder [28] showed that if a Datalog program has the *polynomial fringe property*, which says that every fact in the output has a proof tree of polynomial size, evaluation is in NC. Every linear Datalog program has the polynomial fringe property and is thus in NC. Afrati and Papadimitriou [3] showed that for simple chain queries (including non-linear queries) evaluation is either in NC or P-complete. Recently, Afrati and Ullman [5] studied the tradeoff between communication and number of rounds. They describe a very restricted class of Datalog programs where it is possible to reduce the number of recursion steps (to a number that is logarithmic in the size of the input) without significantly increasing the communication cost.

2.2 Decomposability

The concept of predicate decomposability was first introduced by Wolfson and Silberschatz [32]. A predicate T is decomposable if there are $r > 1$ restricted copies $P_1, P_2, \ldots, P_r$ of the Datalog program P (using arithmetic predicates) such that (i) the copies compute a partition of T for every input, and (ii) there exists an input instance where each copy will produce some input for T. The main result is that decomposability is equivalent to pivoting for sirups where there are no constants, no repeating variables, and the sirup is linear or a simple chain rule. Here, a *sirup* is a Datalog program with one IDB predicate S and two rules: (i) a base rule $S(\boldsymbol{x}) \leftarrow B(\boldsymbol{x})$, and (ii) a recursive rule with head predicate S. A sirup is *linear* if S appears exactly once in the body of the recursive rule.

Later works [30, 31] redefine the concept of decomposability semantically. A Datalog program is *decomposable* if it is possible to partition the output domain (to at least two blocks) such that for every instance I, every output fact has a proof tree where all the IDB facts belong in the same partition block. Wolfson and Ozen [31] show that deciding whether a given Datalog program is decomposable is undecidable. Cohen and Wolfson [30] provide necessary and sufficient syntactic conditions for decomposability for sirups where the arity of the IDB predicate is ≤ 2. They also define the notion of *strongly decomposable* sirups, where the partition must guarantee that, for some input, at least two blocks will produce a fact using the recursive rule of the sirup. Following the same line of work, Zhang et al. [34] present a more general framework that constructs partitionings of the rule instantiations.

2.3 Other Parallel Schemes

In addition to decomposability, several frameworks for parallel recursive processing were introduced in the early 90s [30, 13, 31]. Wolfson [30] generalizes decomposability to *load sharing schemes*, by allowing the output of a predicate to have overlap in the copies of the program P. Under a load sharing scheme, every linear program can be parallelized, even if it is not pivoting. In [13, 14, 31], general schemes are introduced that parallelize the evaluation by partitioning the set of rule instantiations, and allowing for communication among the machines (decomposable and load sharing schemes need no communication). Dewan et al. [12] proposes similar techniques with dynamic adjustments, to balance the load

of a computation. Our framework differs in that the set of rule instantiations is distributed implicitly among the servers, according to the production and consumption policies, and that the communication between servers is made explicit.

2.4 Systems

Recent work studies the implementation of Datalog (or fragments of Datalog) on modern shared-nothing distributed systems. Seo et al. [24] present a distributed version of a Datalog variant for social network analysis called Socialite; however, their framework requires that the user provides annotations to guide the distribution of data. Wang et al. [29] implement a variant of Datalog on the Myria system [17], focusing mostly on asynchronous evaluation and fault-tolerance. The BigDatalog system [26] describes an implementation of Datalog on Apache Spark, but focuses mostly on linear Datalog programs that use aggregation. The task of parallelizing Datalog has also been studied in the context of the popular MapReduce framework [2, 5, 25]. Motik et al. [22] provide an implementation of parallel Datalog in main-memory multicore systems.

3 Preliminaries

We assume an infinite domain **dom**. A database schema σ is a finite set of relation names $\{R_i\}_{i=1}^{n}$ with associated arities $ar(R_i)$. We shall write $R^{(k)}$ to denote a relation R with arity k. A fact $R(a_1, \ldots, a_k)$ over $U \subseteq \mathbf{dom}$ is a tuple consisting of a relation name and a sequence of values from U. We say that $R(a_1, \ldots, a_k)$ is *over* schema σ, if $R^{(k)} \in \sigma$. For a universe $U \subseteq \mathbf{dom}$ and schema σ, we denote by $\mathsf{facts}(\sigma, U)$ the complete set of facts over σ and U. An *instance* I over σ and U is defined as a finite subset of $\mathsf{facts}(\sigma, U)$. We write $I_{|\sigma}$ to denote the subset of I containing all facts in I that are over schema σ.

For $i \in \mathbb{N}$, we abbreviate the set $\{1, \ldots, i\}$ by $[i]$.

3.1 Datalog

We assume an infinite domain of variables **var**, disjoint from **dom**. An *atom* is a formula $R(t_1, \ldots, t_k)$ consisting of a relation name and a tuple of terms; a *term* t_i is either a variable from **var** or a constant from **dom**.

A *Datalog rule* τ is of the form $R(\boldsymbol{x}) \leftarrow S_1(\boldsymbol{y_1}), \ldots, S_n(\boldsymbol{y_n})$, where $R(\boldsymbol{x})$ is a single atom called the head of τ, denoted $head_\tau$, and all $S_i(\boldsymbol{y_i})$ are atoms called body atoms of τ, denoted $body_\tau$. We say that $S_i(\boldsymbol{y_i})$ is *over* schema σ, when $S_i \in \sigma$ and $k = ar(S_i)$. We say that τ is over schema σ if all its atoms are. We assume that Datalog rules are always *safe*, i.e., that all variables in the head occur in at least one body atom. By $vars(\tau)$ we denote the set of variables in rule τ.

A *Datalog program* P is a finite set of Datalog rules. A program P is said to be over schema σ if all its rules are. Particularly, by $\textsc{edb}(P) \subseteq \sigma$ we denote the relation names occurring only in the body of rules, and by $\textsc{idb}(P) \subseteq \sigma$ all other relation names occurring in P. We further distinguish the names in $\textsc{idb}(P)$ by calling some of them *output relations*, denoted $out(P) \subseteq \textsc{idb}(P)$; all other IDB relations are *auxiliary*. We write $\sigma(P)$ to denote $\textsc{edb}(P) \cup \textsc{idb}(P)$.

Consider the directed graph where each node is an IDB predicate, and there is an edge from S to S' if S' occurs in the head of some rule τ of P, and S in the body of τ. We say that P is *recursive* if the graph is cyclic; otherwise, we say it is *non-recursive*. A non-recursive Datalog program with only one rule is called a *conjunctive query* (CQ).

3.2 Evaluation Semantics

We define the evaluation semantics of Datalog programs as usual, through the immediate consequence operator. Let P be a Datalog program and I an instance over $\text{EDB}(P)$. A *valuation* v for rule $\tau \in P$ is a constant-preserving mapping of the terms in τ to values in $\mathbf{dom}$. For a rule $\tau \in P$ and valuation v, we say that τ derives fact $v(head_\tau)$ over instance I if $v(body_\tau) \subseteq I$. We refer to $v(\tau)$ as the instantiation of rule τ with valuation v.

We use T_P to denote the *immediate consequence operator* for P, which applies all rules in P exactly once over a given instance and adds all derived facts to that instance. Formally, $T_P(I) = I \cup \{v(head_\tau) \mid \tau \in P, \text{valuation } v \text{ s.t. } v(body_\tau) \subseteq I\}$. Then, $P(I)$ is defined as the fixpoint reached after iteratively applying the immediate consequence operator over I. It is not difficult to see that T_P is monotone, and thus always reaches a fixpoint after a finite number of iterations. Moreover, the output of the query that P computes is defined as $P(I)_{|out(P)}$. We refer to Abiteboul et al. [1] for a detailed description.

We call a fact $\boldsymbol{f}$ *P-derivable* if $\boldsymbol{f} \in P(I)$ for some instance I, and *P-consumable* if during the evaluation of P on some instance I a rule instantiation $v(\tau)$ fires that requires $\boldsymbol{f}$. Both notions naturally generalize to atoms and predicate symbols, e.g., predicate symbol R is said to be P-consumable if some P-consumable fact $\boldsymbol{f}$ exists with symbol R. Atom A is P-consumable if a rule instantiation as above exists, with $A \in body_\tau$.

3.3 Proof Theoretic Concepts

Let $T = (V, E)$ be a tree. By $fringe_T$ we denote its leaves and by $root_T$ its root vertex. All other vertices are called internal vertices. For a vertex $n \in V$ we denote by $children_T(n)$ the set of child vertices of n in T. We now recall the classical notion of proof tree [1]. A *proof tree* T for a fact $\boldsymbol{f}$ on instance I and Datalog program P is a tree T with vertices over $\mathsf{facts}(\sigma(P), \mathbf{dom})$, where $fringe_T \subseteq I$, $root_T = \boldsymbol{f}$, and for every internal vertex $\boldsymbol{g}$, there is a rule $\tau \in P$ and valuation v such that $\boldsymbol{g} = v(head_\tau)$ and $children_T(\boldsymbol{g}) = v(body_\tau)$. In this case, we shall say that T *uses* the instantiation of τ with valuation v. It is easy to see that $P(I)$ consists of exactly those facts $\boldsymbol{f}$ for which a proof tree for $\boldsymbol{f}$ on I and P exists. We say that a rule instantiation $v(\tau)$ is *useless* if $v(head_\tau) \in v(body_\tau)$; otherwise, we say that it is *useful*. W.l.o.g. we will consider only proof trees where all rule instantiations are useful.

We say that a proof tree T' is *entailed* by proof tree T for P, denoted $T' \sqsubseteq T$, if $fringe_{T'} \subseteq fringe_T$ and $root_{T'} = root_T$.

4 The Framework

Our framework considers a setting with p *servers* (or *machines*) that share no memory and can communicate only via messages – this is commonly referred to as a *shared-nothing* parallel architecture. The set of servers forms a *network* $[p]$ that we assume is fully connected. In order to define how computation is performed, we will use policies that specify how the data (input and output facts) are distributed over the network. We borrow the definition of a distribution policy from [6]:

▶ **Definition 4.1** (Distribution Policy). A *distribution policy* $\boldsymbol{P} = (U, \mathsf{facts}_{\boldsymbol{P}})$ over schema σ and network $[p]$ consists of a universe $U \subseteq \mathbf{dom}$ and a function $\mathsf{facts}_{\boldsymbol{P}} : [p] \to 2^{\mathsf{facts}(\sigma, U)}$ mapping servers to sets of facts over U and σ.

Distribution policies are instance independent, *i.e.*, they are oblivious of the specific database instance. Intuitively, a policy expresses on which servers a fact should reside if

the fact is in the network, but not whether the fact is in the network. Henceforth, we slightly abuse notation and write $\boldsymbol{P}(\boldsymbol{f})$ to denote the set of servers *responsible* for $\boldsymbol{f}$, *i.e.*, $\boldsymbol{P}(\boldsymbol{f}) = \{i \mid \boldsymbol{f} \in \mathsf{facts}_{\boldsymbol{P}}(i)\}$.

In contrast to [6], where the focus is on single-round query evaluation and policies that define only the initial data distribution over EDB facts, we consider a multi-round setting that allows the communication of intermediate facts.

▶ **Definition 4.2** (Economic Policy)**.** An *economic policy* $\boldsymbol{E}$ over schema σ and network $[p]$ is a pair $(\boldsymbol{P}, \boldsymbol{C})$ of distribution policies over the same universe U, where:

- $\boldsymbol{P}$ is defined over $\text{IDB}(P)$ and is called the *production policy*; and
- $\boldsymbol{C}$ is defined over $\text{EDB}(P) \cup \text{IDB}(P)$ and is called the *consumption policy*.

A production policy describes which machines have the responsibility of producing a certain IDB fact. A consumption policy describes which machines need an EDB or IDB fact to satisfy the body of a rule instantiation. We sometimes make universe U explicit, by writing $(\boldsymbol{P}, \boldsymbol{C}; U)$ rather than $(\boldsymbol{P}, \boldsymbol{C})$.[2] We say that a fact $\boldsymbol{f}$ is $\boldsymbol{C}$*-consumable* if $\boldsymbol{C}(\boldsymbol{f}) \neq \emptyset$.

A *family* of economic policies $\mathcal{F}$ is a set of economic policies over a common universe and schema. We say that a family $\mathcal{F}$ satisfies property $\mathcal{P}$ if all the policies in $\mathcal{F}$ satisfy $\mathcal{P}$.

4.1 Datalog Evaluation Modulo Policies

Instead of letting a server compute the full program over its local instance, we restrict the evaluation process based on a server's economic policy. That is, for economic policy $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ and Datalog program P, the following sequential evaluation algorithm takes place on server i:

- First, every rule $\tau \in P$ is annotated with policy-predicates as follows. For the head $R(\boldsymbol{x})$, we add a predicate $\mathit{Policy}_R(\boldsymbol{x})$ to the body of τ. Here, predicate Policy_R refers to relation $\mathsf{facts}_{\boldsymbol{P}}(i)_{|\{R\}}$.
- Second, for every atom $S(\boldsymbol{y})$ in the body of τ, we add the predicate $\mathit{Policy}_S(\boldsymbol{y})$, where now Policy_S refers to the relation $\mathsf{facts}_{\boldsymbol{C}}(i)_{|\{S\}}$.

The added predicates may be infinitely large, but can be accessed through queries of the form "$\boldsymbol{t} \in \mathsf{facts}_{\boldsymbol{P}}(i)_{|\{R\}}$?" or "$\boldsymbol{t} \in \mathsf{facts}_{\boldsymbol{C}}(i)_{|\{S\}}$?".

Throughout the paper we assume the semi-naive evaluation strategy for Datalog programs. Semi-naive evaluation proceeds as usual over the annotated program: after each application of the fixpoint operator, the newly derived facts are added to a delta relation, and a rule instantiation is triggered only if at least one of its facts is in the delta relation from the previous iteration. We denote by $P_{\restriction \boldsymbol{E}}(I, J)$ the fixpoint instance when we execute P restricted to $\boldsymbol{E}$ on input I, with delta relations initialized with J.

4.2 Distributed Evaluation Strategy

We now present how an economic policy induces a parallel evaluation strategy. Our parallel model is the BSP-based Massively Parallel Communication Model (MPC) [21]. In this model, computation is performed over servers in a multi-round fashion. Each round has two distinct phases: a local computation phase, and a synchronized communication phase.

Consider a Datalog program P, a network $[p]$, and an economic policy $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$. Moreover, let I be the input instance, which we initially assume to be partitioned arbitrarily

[2] Notice that mentioning U is redundant, but allows a slightly simpler notation, since $\boldsymbol{P}$ and $\boldsymbol{C}$ need not be specified explicitly to reference their universe U.

over the p servers. Denote by I_i the initial local instance of machine i. Let local_i^k be the instance on machine i right after the k-th communication phase.

We consider the following procedure: Initially, we set $\mathsf{local}_i^0 \leftarrow I_i$. Then, at the k-th round (for $k \geq 1$), we perform the following:

1. *Communication:* Every machine sends its facts as defined by the consumption policy $\boldsymbol{C}$. That is, server i sends local fact $\boldsymbol{f} \in \mathsf{local}_i^{k-1}$ to server j if (and only if) $\boldsymbol{f} \in \mathsf{facts}_{\boldsymbol{C}}(j)$. Let rec_i^k be the facts received by machine i during the k-th communication phase.[3]
2. *Computation:* Every server computes the local fixpoint: if $k = 1$, then $\mathsf{local}_i^k = P_{\restriction \boldsymbol{E}}(\mathsf{rec}_i^k \cup \mathsf{local}_i^{k-1}, \mathsf{rec}_i^k \cup \mathsf{local}_i^{k-1})$, otherwise $\mathsf{local}_i^k = P_{\restriction \boldsymbol{E}}(\mathsf{rec}_i^k \cup \mathsf{local}_i^{k-1}, \mathsf{rec}_i^k \setminus \mathsf{local}_i^{k-1})$.

Intuitively, the algorithm terminates when, after a round is finished, for every server all locally derived facts that need to be sent to some other server according to the consumption policy, were already sent to these servers in an earlier round.

Formally, for server i, we define set $F_i = \{\boldsymbol{f} \mid \boldsymbol{C}(\boldsymbol{f}) \setminus i \neq \emptyset\}$. Intuitively, F_i represents all facts consumed by servers other than i itself. We say that a server has reached a *local fixpoint state* for $\boldsymbol{E}$ and P after round $k \geq 1$, if $\mathsf{local}_i^k \cap F_i \subseteq \mathsf{local}_i^{k-1}$. We say that the network $[p]$ has reached a *global fixpoint state* for $\boldsymbol{E}$ and P after round k, if all servers $i \in [p]$ have reached a local fixpoint state after round k. Notice that this condition is as desired, because every round goes into the communication phase first, then into the local computation phase.

One could imagine a smarter communication procedure that incorporates Datalog semantics as well. For example, a server does not need to send a local fact $\boldsymbol{f} \in \mathsf{facts}_{\boldsymbol{C}}(j)$ to server j if for every input I server j is guaranteed to already have $\boldsymbol{f}$ in its local instance. However, it is in general undecidable to make such a decision (see Lemma 5.3).

For instance I, let $[P, \boldsymbol{E}](I)$ denote the union of all facts over $out(P)$ found at any server after reaching the global fixpoint. Notice that the above evaluation strategy always reaches a fixpoint, due to monotonicity of Datalog.

▶ **Example 4.3.** Consider the left-linear Datalog program that computes transitive closure:

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

For any function $h : \mathbf{dom} \to [p]$, we define the economic policy $(\boldsymbol{P}_1, \boldsymbol{C}_1)$, where $\boldsymbol{C}_1(R(a, b)) = [p]$, and $\boldsymbol{C}_1(T(a, b)) = \boldsymbol{P}_1(T(a, b)) = \{h(a)\}$ for all $a, b \in \mathbf{dom}$. This policy works as follows: it replicates the EDB facts everywhere, and then produces/consumes each fact $T(a, b)$ at machine $h(a)$. It is easy to see that the economic policy correctly computes the transitive closure. In fact, the evaluation always terminates in a single round.

Consider a different policy $(\boldsymbol{P}_2, \boldsymbol{C}_2)$, which again takes any function $h : \mathbf{dom} \to [p]$ and has $\boldsymbol{C}_2(R(a, b)) = \{h(a)\}$, $\boldsymbol{C}_2(T(a, b)) = \{h(b)\}$, and $\boldsymbol{P}_2(T(a, b)) = [p]$. This policy does not replicate the EDB facts, but it hash-partitions them according to the first attribute. Whenever a machine discovers a new fact, the new fact has to be consumed to the location determined by the hash of the second attribute. Observe that the production policy is $[p]$ because we do not know where each fact will be produced (in other words, each machine will produce as many IDB facts as possible from its local input without any restrictions).

We will see later in Section 6 that all the above economic policies belong in a specific family of policies that we call Generalized Hypercube Policies (GHPs). We notice that our framework supports evaluation strategies that are *oblivious* of the instance: each fact is

[3] We remark that from a practical viewpoint it makes no sense to communicate the same facts more than once. When $j = i$, no actual communication takes place.

communicated, consumed, and produced independent of whether other facts are in the same local instance or not. Lastly, we note that monotonicity of Datalog ensures monotonicity of economic policies for Datalog Programs.

▶ **Proposition 4.4.** *For every Datalog program P and economic policy $\boldsymbol{E}$ for P, $\boldsymbol{f} \in [P, \boldsymbol{E}](I')$ implies $\boldsymbol{f} \in [P, \boldsymbol{E}](I)$, for all $I' \subseteq I$. More specifically, if $\boldsymbol{f}$ is derived by $\boldsymbol{E}$ for I' in round i on server s, then $\boldsymbol{f}$ is derived by $\boldsymbol{E}$ for I in round $j \leq i$ on server s.*

5 Parallel-Correctness

An economic policy for a Datalog program does not necessarily lead to the desired output. For example, if the production policy maps every fact onto the empty set of servers, then the execution will generate only empty IDB relations. Henceforth, we are only interested in economic policies that generate the expected output.

▶ **Definition 5.1** (Parallel-correctness)**.** An economic policy $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C}; U)$ is *parallel-correct* for Datalog program P if $[P, \boldsymbol{E}](I) = P(I)|_{out(P)}$, for every instance $I \subseteq \mathsf{facts}(\text{EDB}(P), U)$.

Parallel-correctness is in general undecidable, even for simple classes of policies. For instance, consider the class of policies, where $\boldsymbol{P}(\boldsymbol{f}_1) = \boldsymbol{P}(\boldsymbol{f}_2)$ and $\boldsymbol{C}(\boldsymbol{f}_1) = \boldsymbol{C}(\boldsymbol{f}_2)$, whenever $\boldsymbol{f}_1, \boldsymbol{f}_2$ are facts with same relation symbol. We call this class of policies *value-independent*, denoted $\mathcal{E}_{indep}$, since the facts are mapped to machines only according to the relations they belong to. Value-independent policies allow a succinct representation by simply enumerating the IDB predicates of P and the subsets of $[p]$ where each relation is assigned.

We consider the following decision problem.

	PC$(\mathcal{L}, \mathcal{E})$
Input:	Program $P \in \mathcal{L}$, policy $\boldsymbol{E} \in \mathcal{E}$.
Question:	Is $\boldsymbol{E}$ parallel-correct for P?

▶ **Theorem 5.2.** PC$(Datalog, \mathcal{E}_{indep})$ *is undecidable.*

The proof is given in Appendix A.1. We next show an even stronger result:

▶ **Lemma 5.3.** *Let P be an arbitrary Datalog program and $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C}; U)$ an economic policy over σ that is parallel-correct for P. Now let $\boldsymbol{f} \in \mathsf{facts}(\sigma, U)$, and $\boldsymbol{C}'$ the consumption policy where $\boldsymbol{C}'(\boldsymbol{g}) = \boldsymbol{C}(\boldsymbol{g})$ for all $\boldsymbol{g} \in \mathsf{facts}(\sigma, U) \setminus \{\boldsymbol{f}\}$ and $\boldsymbol{C}'(\boldsymbol{f}) \subsetneq \boldsymbol{C}(\boldsymbol{f})$. It is still undecidable whether $\boldsymbol{E}'$ is parallel-correct for P.*

Despite the above results, we can present some syntactic conditions that are necessary for parallel-correctness, and some that are sufficient.

▶ **Definition 5.4** (Support)**.** An instantiation of rule τ with valuation v is *supported* by economic policy $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ if there exists some machine $s \in [p]$ with $v(head_\tau) \in \mathsf{facts}_{\boldsymbol{P}}(s)$ and $v(body_\tau) \subseteq \mathsf{facts}_{\boldsymbol{C}}(s)$.

We consider various categories of economic policies based on which rule instantiations are supported for a given Datalog program P:

$\mathcal{N}_P^{all}$: the set of all rule instantiations of P.

$\mathcal{N}_P^{min}$: the set of all minimal rule instantiations of P. An instantiation of rule τ with valuation v is *minimal* if there is no rule τ' and valuation v' with $v'(head_{\tau'}) = v(head_\tau)$ and $v'(body_{\tau'}) \subsetneq v(body_\tau)$.

$\mathcal{N}_P^{use}$: the set of all rule instantiations of P that are useful. Recall that an instantiation of rule τ with valuation v is useful if $v(head_\tau) \notin v(body_\tau)$.

$\mathcal{N}_P^{ess}$: the set of all essential rule instantiations of P. An instantiation of rule τ with valuation v is *essential* if for some P-derivable fact $\boldsymbol{f}$ and instance I, every proof tree T for $\boldsymbol{f}$ on I and P has a vertex $\boldsymbol{g}$ with $\boldsymbol{g} = v(head_\tau)$ and $v(body_\tau) \subseteq children_T(\boldsymbol{g})$.

If the program is non-recursive, then $\mathcal{N}_P^{use} = \mathcal{N}_P^{all}$, since there will be no rule that contains the same relation in the head and the body. We also have:

▶ **Proposition 5.5.** *For every Datalog program P, we have $\mathcal{N}_P^{ess} \subseteq \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$.*

The proof is in Appendix A.2. The following example demonstrates the different types of rule instantiations.

▶ **Example 5.6.** Let P be the left-linear transitive closure program from Example 4.3; consider a rule instantiation of the recursive rule: $T(a,b) \leftarrow T(a,c), R(c,b)$, for some (not necessarily different) constants a, b, c. We distinguish the following cases:

$c = a$: in this case, the instantiation is not minimal, since we can derive the same head fact from the instantiation $T(a,b) \leftarrow R(a,b)$ of the first rule.

$c = b$: in this case, the instantiation is useless, since $T(a,b)$ also belongs in the body. In some sense, this derivation is unnecessary, as we have already "discovered" the head fact.

$c \neq a, c \neq b$: in this case, the instantiation is minimal and useful; it is also essential. To show this, consider the instance $I = \{R(a,c), R(c,b)\}$, and the fact $\boldsymbol{f} = T(a,b)$. Because $c \notin \{a,b\}$, the only proof tree for $\boldsymbol{f}$ without "useless" rule instantiations is the one with root $\boldsymbol{f}$, children $T(a,c), R(c,b)$, where $T(a,c)$ has $R(a,c)$ as child.

Depending on which types of rule instantiations are supported by an economic policy, we can define different types of policies. An economic policy that supports all possible rule instantiations, that is, $\mathcal{N}_P^{all}$, is said to be *strongly supporting* for Datalog program P.

▶ **Proposition 5.7.** *Let P be a Datalog program and $\boldsymbol{E}$ an economic policy. If $\boldsymbol{E}$ supports all minimal and useful rule instantiations in P, then it is parallel-correct. If $\boldsymbol{E}$ is parallel-correct for P, then it supports all essential rule instantiations.*

▶ **Proposition 5.8.** *Let P be a Datalog program where each* IDB *predicate occurs only in the head of rules (i.e., P is a union of CQs). Then, $\mathcal{N}_P^{ess} = \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$.*

Proofs are given in Appendix A.3 and A.4.

Together with Proposition 5.7, the above proposition implies that a Datalog program where the body of each rule contains only EDB relations is parallel-correct if and only if it supports every minimal rule instantiation, or equivalently if and only it supports every essential rule instantiation. Notice that this class of Datalog programs corresponds to a program that computes a set of UCQs, and thus the above result captures the characterization of parallel-correctness for CQs and UCQs in [6, 15]. We should emphasize here that [6, 15] consider only economic policies where $\boldsymbol{P}$ assigns every fact to every server, while a general economic policy can assign facts to any subset of servers.

For general Datalog programs, $\mathcal{N}_P^{ess} = \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$ is not true anymore, and thus supporting essential instantiations is not a sufficient condition for parallel-correctness, even if P is non-recursive. (Recall that non-recursiveness is a syntactic condition, and that all such programs are straightforwardly rewritable to UCQs.)

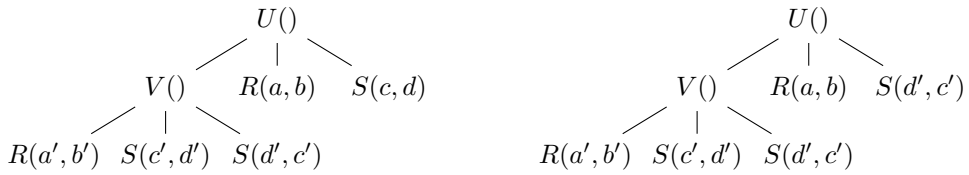

Figure 1 The two proof trees for the fact $\boldsymbol{f} = U(a, b)$.

▶ **Example 5.9.** Consider the following non-recursive Datalog program P:

$$V() \leftarrow R(x, y), S(z, w), S(w, z). \qquad U() \leftarrow V(), R(x, y), S(z, w).$$

and take the rule instantiation with head $U()$ and body $\{V(), R(a, b), S(c, d)\}$. Assume that $c \neq d$. This rule instantiation is minimal, but we will show that it is not essential.

For the sake of contradiction, assume that it is essential. Then, for some instance I there exists a proof tree T for $U()$ on I and P such that there exists a vertex $U()$ with $\{V(), R(a, b), S(c, d)\} \subseteq \mathit{children}_T(U())$. Since the proof tree contains the fact $V()$, it also contains a rule instantiation that derives the fact $V()$ with body $\{R(a', b'), S(c', d'), S(d', c')\}$ for some constants a', b', c', d'. We can now construct two proof trees for $U()$ on the same instance, as seen in Figure 1. Because $c \neq d$, one of the facts $S(c', d'), S(d', c')$ must be different from $S(c, d)$ (In Figure 1 we assume this fact is $S(d', c')$). Thus, for one of the two trees, the children of $U()$ will not be a subset of $\{V(), R(a, b), S(c, d)\}$. This implies that the rule instantiation we considered is indeed not essential.

▶ **Example 5.10.** This example shows that $\mathcal{N}_P^{ess} = \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$ can hold for recursive programs. Consider Example 5.6. Notice that every rule instantiation of the base rule, $T(x, y) \leftarrow R(x, y)$, is trivially minimal, useful and essential. As for the recursive rule, we showed in Example 5.6 that an instantiation that is minimal and useful is also essential. Observe that if this instantiation is only minimal but not useful, or only useful and not minimal, it is not essential. Thus, both properties are necessary to guarantee essentiality.

We conclude this section by commenting on whether it is computationally feasible to test the different properties of rule instantiations. It is easy to see that given an instantiation, it is possible to check whether it is useful in polynomial time. The complexity for checking the minimality of a rule instantiation is coNP-complete [6]. Unfortunately, testing essentiality of a rule instantiation is undecidable.

▶ **Proposition 5.11.** *Testing essentiality of rule instantiations is undecidable.*

6 Generalized Hypercube Policies

In this section, we present a general class of economic policies, called *Generalized Hypercube Policies (GHP)*, which encompass a broad variety of evaluation strategies.

We first give an intuitive explanation. The formalism of GHPs relies on the Hypercube partitioning for CQs [4], which has been shown to provide nice guarantees on the communication-cost for CQ evaluation [7]. Let $P = \{\tau\}$ be a CQ with k distinct variables. Hypercube conceptually orders the p servers as a hypercube $\mathcal{P} = [p_1] \times [p_2] \times \cdots \times [p_k]$, with $\prod_i p_i = p$, where every dimension $p_i \geq 0$ corresponds to a variable x_i from the query; every server is assigned a unique coordinate in space $\mathcal{P}$; and every variable x_i is associated to a hash function $h_{x_i} : \mathbf{dom} \mapsto [p_i]$. Then, a fact $R(a_1, \ldots, a_r)$, matching with atom

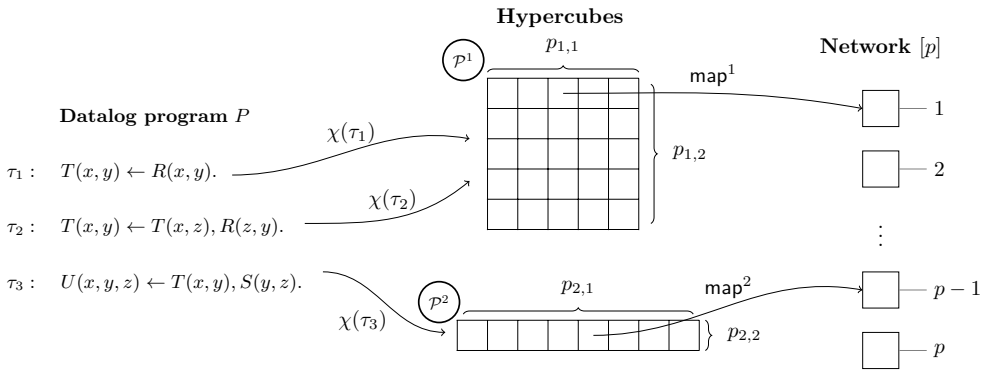

Figure 2 Example of a GHP policy for the Datalog program P with three rules.

$R(y_1, \ldots, y_r) \in body_\tau$, is sent to all servers whose coordinate agrees, for all $j \in [r]$, with position $h_{y_j}(a_j)$ on the dimension of $\mathcal{P}$ where y_j is associated with.

For GHPs we associate to every rule a hypercube over the full p-server network, and intuitively define the consumption policy so that "a fact is consumed at server i if and only if one of the considered Hypercube specifications would send it to server i"; for the specification of the production policy, we rely on a similar mechanism.

GHP parameters

Let P be a Datalog program, and assume we have a network $[p]$. A *GHP for* P defines a finite set of k-dimensional *hypercubes* $\mathcal{P}^1, \ldots, \mathcal{P}^\ell$, for some parameter k.[4] The size of the dimensions of the hypercubes are parametrized by a matrix $\mathbf{P}$ of dimensions $\ell \times k$, such that $\prod_{i=1}^{k} p_{j,i} = p$, for each $j \in [\ell]$. Each hypercube is then defined as $\mathcal{P}^j = [p_{j,1}] \times [p_{j,2}] \times \ldots [p_{j,k}]$. For each hypercube $\mathcal{P}^j$, we also define a bijective mapping map^j that assigns to every point in $\mathcal{P}^j$ a machine $s \in [p]$. The latter thus provides the mapping between conceptual machines in the cube and real machines in the considered network.

A GHP policy next assigns each rule τ to exactly one of the hypercubes: let $\chi : P \to [\ell]$ be the function that encodes this assignment. Given this assignment, a GHP defines a mapping $\rho^\tau : [k] \to 2^{vars(\tau)}$ that maps each dimension of the hypercube $\mathcal{P}^{\chi(\tau)}$ to a subset of the variables that appear in τ, such that the following condition holds:

> If $\chi(\tau) = \chi(\tau')$, then $|\rho^\tau(i)| = |\rho^{\tau'}(i)|$ for every dimension i. In other words, the mappings of variables of different rules to the same hypercube must be consistent.

Finally, the GHP defines for each dimension $i \in [k]$ and each hypercube $\mathcal{P}^j$ a *hash function* h_i^j that maps sets of size $\leq \alpha$ (α is the size of the set $\rho^\tau(i)$ for any τ such that $\chi(\tau) = j$) to a value in the i-th dimension. We require hash functions to be surjective. Notice that our concept of hash-function is a generalization of the hash-functions used in, e.g., the Hypercube algorithm, where $\alpha = 1$. Further, we notice that, by definition, rules that use the same hypercube, also use the same hash function for each dimension of that hypercube.

[4] We assume w.l.o.g. that each hypercube has the same number of dimensions, but we can also define it such that different rules have a different number of dimensions.

GHP semantics

Let $\boldsymbol{f}$ be a fact and suppose that $\boldsymbol{f} = v(A)$, for some valuation v and atom $A = R(\boldsymbol{y})$ that appears in rule τ.[5] We define the following set of machines:

$$S^{\tau}_{\boldsymbol{f},A} = \{\mathsf{map}^{\chi(\tau)}(\mathbf{q}) \mid \mathbf{q} \in \mathcal{P}^{\chi(\tau)} \text{ such that } \forall i \text{ with } \emptyset \subsetneq \rho^{\tau}(i) \subseteq \boldsymbol{y} : \mathbf{q}_i = h_i^{\chi(\tau)}(v(\rho^{\tau}(i)))\}.$$

Intuitively, $S^{\tau}_{\boldsymbol{f},A}$ denotes the set of machines whose coordinate $\boldsymbol{q}$ is consistent with the hash mappings specified for τ. Notice that if the atom $R(\boldsymbol{y})$ has only a part of the variables that correspond to some dimension i, then facts are broadcast over dimension i, as it happens if none of these variables are in $\boldsymbol{y}$.

The consumption policy $\boldsymbol{C}(\boldsymbol{f})$ is defined as the union over all sets $S^{\tau}_{\boldsymbol{f},A}$ for rules τ and atoms $A \in \mathit{body}_{\tau}$ with instantiation $\boldsymbol{f}$. The production policy $\boldsymbol{P}(\boldsymbol{f})$ is similarly defined as the union over all sets $S^{\tau}_{\boldsymbol{f},A}$ for rules τ and atom head_{τ} with instantiation $\boldsymbol{f}$.

▶ **Example 6.1.** Consider the Datalog program depicted in Figure 2. We choose two hypercubes $\mathcal{P}^1, \mathcal{P}^2$ ($\ell = 2$) with dimension $k = 2$. The first two rules τ_1, τ_2 are mapped to the hypercube $\mathcal{P}^1$, and the third rule τ_3 is mapped to $\mathcal{P}^2$. We choose the dimensions of the hypercubes such that $p_{1,1} \cdot p_{1,2} = p$, $p_{2,1} = p$, and $p_{2,2} = 1$. The two functions $\mathsf{map}^1, \mathsf{map}^2$ map the points of $\mathcal{P}^1, \mathcal{P}^2$ respectively to $\{1, \ldots, p\}$ in a one-to-one fashion. Finally, the mapping of variables to dimensions is:

$$\rho^{\tau_1}(1) = \{x\}, \rho^{\tau_1}(2) = \{y\}, \rho^{\tau_2}(1) = \{x\}, \rho^{\tau_2}(2) = \{z\}, \rho^{\tau_3}(1) = \{y\}, \rho^{\tau_3}(2) = \{\}$$

Consider the first two rules (which form the left-linear TC example), and assume that $p_{1,1} = 1$ and $p_{1,2} = p$. Then, the resulting GHP is equivalent to the hash partitioning policy that we described in Example 4.3. Notice that since we use the same hypercube for both rules, the EDB relation R will be hash partitioned only once. If we now change the dimensions to $p_{1,1} = p, p_{1,2} = 1$, we obtain the decomposable policy of Example 4.3 that broadcasts the EDB R to every machine and can terminate in a single round. Apart from the above two GHPs, we can also define other GHPs by configuring different dimensions of the hypercube $\mathcal{P}^1$. For example, we can choose $p_{1,1} = p_{1,2} = \sqrt{p}$.

We next show that GHPs are strongly supporting policies. A proof is in Appendix A.5.

▶ **Proposition 6.2.** *Let P be a Datalog program. Every GHP $\boldsymbol{E}$ for P is strongly supporting for P and, as a consequence, parallel-correct for P.*

GHP Families

Since we do not want to consider an encoding mechanism for hash functions – which is necessary to formally reason about properties for GHPs – we introduce the concept of GHP families. Given a Datalog program P and network $[p]$, a *GHP family* $\mathcal{H}$ is defined as the set of GHPs over P and $[p]$ that all have the same parametrization for $\mathbf{P}, \mathsf{map}^j, \chi, \rho^{\tau}$. In other words, policies in $\mathcal{H}$ can differ only with respect to the choice of hash functions, and for every choice of hash functions, the associated GHP is in the family. By $\mathcal{F}_{\text{GHP}}$ we denote the class of all GHP families.

[5] Notice that either v does not exist, or is unique for the variables in atom A.

7 Bounded & Disjoint Evaluation

In this section, we we ask two main questions: First, can we reason about the number of rounds that an economic policy needs to compute a Datalog program? Second, can we constrain the number of machines that derive a copy of the same fact? We start with a formal definition of boundedness.

▶ **Definition 7.1** (Boundedness). An economic policy $\boldsymbol{E}$ for Datalog program P is *bounded* if some constant k exists such that, for every instance I, the network reaches a global fixpoint for $\boldsymbol{E}$ and P, when round k is finished. We say $\boldsymbol{E}$ is ℓ-bounded if $k \leq \ell$.

One should not confuse the number of rounds in the parallel computation with the number of iterations of semi-naive evaluation. Nevertheless, as the following proposition shows, boundedness of the Datalog program implies boundedness of the evaluation.

▶ **Proposition 7.2.** *If P is a bounded Datalog program, then every parallel-correct economic policy $\boldsymbol{E}$ for P is k-bounded, for some constant k that depends on P.*

Surprisingly, there exist economic policies for bounded Datalog programs that are not bounded. However, due to Proposition 7.2, such policies cannot be parallel-correct.

▶ **Example 7.3.** Consider the following bounded program.

$$T(x) \leftarrow A(x). \qquad T(x) \leftarrow B(x), T(y).$$

We construct a network with $p > 1$ machines. Consider a policy that consumes $T(i)$ and $B(i)$ at machine $(i \mod p)+1$, and produces $T(i)$ at machine $(i \mod p)$. Every tuple in A is consumed at machine 1. Now, consider the following input instance: $\{A(0), B(1), B(2), \ldots, B(p-1)\}$. It is easy to see that $T(0)$ is produced at machine 1 at round 1, $T(1)$ is produced at machine 2 at round 2, and so on, until $T(p-1)$ is produced at round p at machine p.

In the remainder of this section, we focus on pure Datalog (denoted *PureDatalog*). We call a Datalog program *pure* if its variables occur at most once in every atom and it has no constants [23]. We consider the following decision problems.

k-BOUNDEDNESS$(\mathcal{L}, \mathcal{E})$	
Input:	Program $P \in \mathcal{L}$, policy $\boldsymbol{E} \in \mathcal{E}$.
Question:	Is $\boldsymbol{E}$ k-bounded for P?

BOUNDEDNESS$_F(\mathcal{L}, \mathcal{W})$	
Input:	Program $P \in \mathcal{L}$, family $\mathcal{F} \in \mathcal{W}$.
Question:	Is there a k s.t. $\mathcal{F}$ is k-bounded for P?

k-BOUNDEDNESS$_F(\mathcal{L}, \mathcal{W})$	
Input:	Program $P \in \mathcal{L}$, family $\mathcal{F} \in \mathcal{W}$.
Question:	Is $\mathcal{F}$ k-bounded for P?

▶ **Theorem 7.4.**

1. BOUNDEDNESS$_F(\mathit{PureDatalog}, \mathcal{F}_{\text{GHP}})$ *is undecidable;*
2. k-BOUNDEDNESS$(\mathit{PureDatalog}, \mathcal{E}_{indep})$ *and* k-BOUNDEDNESS$_F(\mathit{PureDatalog}, \mathcal{F}_{\text{GHP}})$ *are undecidable for $k \geq 2$; and*
3. k-BOUNDEDNESS$_F(\mathit{PureDatalog}, \mathcal{F}_{\text{GHP}})$ *is in* PTIME *if $k = 1$.*

Result (3) follows from the syntactical characterization shown in the next subsection. Towards this characterization, we first give a general characterization of 1-boundedness for strongly supporting policies.

Let P be a Datalog program and $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ an economic policy. We denote by $\boldsymbol{P}^*$ the policy obtained by removing from every $\boldsymbol{P}(\boldsymbol{f})$ any server s for which no rule instantiation $v(\tau)$ exists with $v(\mathit{head}_\tau) = \boldsymbol{f}$, $v(\mathit{body}_\tau) \subseteq \mathsf{facts}_{\boldsymbol{C}}(s)$, and $v(\mathit{body}_\tau)$ being all P-derivable. Intuitively, $\boldsymbol{P}^*(\boldsymbol{f})$ removes those servers that are allowed to produce $\boldsymbol{f}$, but cannot due to limitations of the consumption policy $\boldsymbol{C}$. Notice that if $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ is strongly supporting for P, then so is $\boldsymbol{E} = (\boldsymbol{P}^*, \boldsymbol{C})$, since we have not removed the support of any rule instantiation.

▶ **Proposition 7.5.** *Let P be a Datalog program and $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ a strongly supporting economic policy for P. $\boldsymbol{E}$ is 1-bounded if and only if for every P-derivable* IDB *fact $\boldsymbol{f}$: (1) $|\boldsymbol{C}(\boldsymbol{f})| \leq 1$; and (2) $|\boldsymbol{C}(\boldsymbol{f})| = 1$ implies $\boldsymbol{C}(\boldsymbol{f}) = \boldsymbol{P}^*(\boldsymbol{f})$.*

7.1 Weakly Pivoting GHPs

We present a necessary and sufficient syntactic condition for 1-boundedness of GHP families. Here, for atom A and set of variables $X \subseteq \mathit{vars}(A)$, we denote by $\mathit{pos}_A(X)$ the positions in A having variables from X.

▶ **Definition 7.6** (Pivoting Relation)**.** A relation R is *pivoting for GHP family $\mathcal{H}$* if for every two atoms A_1, A_2 (in rules τ_1, τ_2 respectively) over R, and for all dimensions i of cube $\chi(\tau_1)$ with $p_{\chi(\tau_1),i} > 1$: $\emptyset \subsetneq \rho^{\tau_1}(i) \subseteq \mathit{vars}(A_1)$; $\chi(\tau_1) = \chi(\tau_2)$; and $\mathit{pos}_{A_1}(\rho^{\tau_1}(i)) = \mathit{pos}_{A_2}(\rho^{\tau_2}(i))$.

Intuitively, if R is pivoting, then every rule that sends R tuples will send each R tuple to exactly one machine, and the rules agree on this machine.

▶ **Example 7.7.** For example, take the program

$$\tau_1 : T(x,y) \leftarrow R(x,y). \qquad \tau_2 : T(x,y) \leftarrow T(x,z), R(z,y). \qquad \tau_3 : O(y) \leftarrow T(x,y), S(x).$$

and the GHP over the single one-dimensional cube (*cube* 1). We define $\chi(\tau_1) = \chi(\tau_2) = \chi(\tau_3) = 1$ and $\rho^{\tau_1}(1) = \rho^{\tau_2}(1) = \rho^{\tau_3}(1) = \{x\}$. Let map^1 be the identity mapping. Here, S and T are pivoting relations; O and R are not pivoting.

▶ **Definition 7.8** (Pivoting/Weakly pivoting)**.** We say that a GHP family is *pivoting* (*weakly pivoting*, resp.) for P if all (all P-consumable, resp.) IDB relations are pivoting.

The program from Example 7.7 is weakly pivoting. We can test whether a GHP family is weakly pivoting in polynomial time, since we need to go over all P-consumable IDB relations, and then for each such relation R test all pairs of atoms over R. This observation, along with the proposition below – that shows that weakly pivoting is a necessary and sufficient condition for 1-boundedness – implies that deciding 1-boundedness for GHP families is indeed in PTIME.

▶ **Proposition 7.9.** *Let P be a pure Datalog program, and $\mathcal{H}$ a GHP family. Then, $\mathcal{H}$ is 1-bounded for P* if and only if *it is weakly pivoting for P.*

We remark that Proposition 7.9 cannot be easily generalized. For example, one cannot replace GHP families by strongly supporting policies, since then facts that are not P-consumable may still be $\boldsymbol{C}$-consumable (*i.e.*, $\boldsymbol{C}(\boldsymbol{f}) \neq \emptyset$). Reasoning about the latter requires a concrete representation mechanism. Further, it is unclear what the complexity becomes for testing 1-boundedness under general (not necessarily pure) Datalog, since then it is required to reason about P-derivability of facts. An example is given in Appendix A.6.

7.2 Weakly Pivoting Datalog

We have so far looked at whether a given GHP family is 1-bounded. In this section, we ask: *which Datalog programs admit a 1-bounded policy?*

If $A = R(\boldsymbol{x})$ is an atom, we use $A[i]$ to denote the variable/constant in atom A in position i. We naturally extend $A[\cdot]$ to map tuples of positions (that take values from the set $\{1, \ldots, ar(R)\}$) onto tuples of variables/constants. For example, if $A = R(x_1, x_2, x_3)$ and $\boldsymbol{b} = (1, 3)$, then $A[\boldsymbol{b}] = (A[1], A[3]) = (x_1, x_3)$.

▶ **Definition 7.10** (Pivot Base)**.** Let P be a Datalog program, and let $\sigma \subseteq \text{IDB}(P)$. Let $\boldsymbol{\beta}$ be a function that takes as input some $R \in \sigma$ and outputs a non-empty tuple with values in $[ar(R)]$. We say that $\boldsymbol{\beta}$ is a *pivot base* for σ if:

- For every rule $\tau \in P$ and for every pair of atoms $R(\boldsymbol{x})$, $S(\boldsymbol{y})$ in $\{head_\tau\} \cup body_\tau$, such that $R, S \in \sigma$, we have $R(\boldsymbol{x})[\boldsymbol{\beta}(R)] = S(\boldsymbol{y})[\boldsymbol{\beta}(S)]$.

A Datalog program P is *pivoting* (*weakly pivoting*, resp.) if it has a pivot base for all relations in $\text{IDB}(P)$ (for all relations in $\text{IDB}(P)$ that occur in the body of some rule in P).

An example is given in Appendix A.7.

The concept pivoting Datalog was first introduced for single rule programs [30] and then generalized to full Datalog [23] where it is called *generalized pivoting*. The latter definition is based on a rather complex argument over fractional weight-mappings, but relates to pivoting in that every generalized pivoting Datalog program is pivoting for *all* IDB relations. For pure Datalog these notions are equivalent. The proposition below shows that for pure Datalog, a weakly pivoting program admits a weakly pivoting (and thus 1-bounded) GHP family.

▶ **Proposition 7.11.** *Let P be a pure Datalog program and $p \geq 2$. There is a 1-bounded GHP family if and only if P is weakly pivoting.*

7.3 Bounded and Disjoint Evaluation

Sometimes we want to guarantee that, at the end of computation, no two copies of the same fact have been derived at different machines. We call this property disjointness.

▶ **Definition 7.12** (Disjointness)**.** Let P be a Datalog program, and R an IDB predicate of P. We call an economic policy $\boldsymbol{D}$ for P *R-disjoint* if for every instance, every fact of R is produced in at most one server.

We study economic policies that are both 1-bounded *and* disjoint. For this, let P be a Datalog program and $\boldsymbol{D}$ a strongly supporting economic policy for P over $[p]$. We call $s \in [p]$ a *straggler* if $s \in \boldsymbol{C}(\boldsymbol{f})$ or $s \in \boldsymbol{P}^*(\boldsymbol{f})$ for all facts $\boldsymbol{f}$ of some IDB relation where P is defined over. Intuitively, a straggler is a server that consumes or produces an entire relation.

▶ **Proposition 7.13.** *Let $P \in PureDatalog$ and $\mathcal{H}$ a GHP family for P. Then, $\mathcal{H}$ is 1-bounded, disjoint for P, and without stragglers for IDB relations, if and only if, $\mathcal{H}$ is pivoting.*

Next, we show which programs admit a 1-bounded, disjoint policy.

▶ **Proposition 7.14.** *Let $P \in PureDatalog$. Then P is pivoting if, and only if, P admits a 1-bounded, strongly supporting, disjoint economic policy without stragglers for IDB relations.*

▶ Remark. The reader may wonder how the above concepts relate to the class of decomposable programs [32, 31]. A *decomposable* program is a (single rule) Datalog program that admits an evaluation strategy (via predicate restrictions) that is parallel-correct, 1-bounded, disjoint,

and non-trivial. (Here non-triviality means that all servers do part of the work.) We did not consider the non-triviality property, but instead require the absence of stragglers. Nevertheless, for GHPs, non-triviality is implied – at least for pure Datalog – by the use of surjective hash functions).

8 Conclusion

We introduce a theoretical framework to reason about multi-round Datalog evaluation in a distributed setting. In this framework we study three properties: parallel-correctness, boundedness, and disjointness. There are many interesting questions left open. For example, it would be interesting to come up with restrictions on Datalog programs and economic policies, for which the mentioned properties are not undecidable. Another interesting direction for future work would be to define a relevant fairness condition for economic policies, e.g., an instance independent notion of load-balancing; and to study bounds on the amount of communication needed to evaluate Datalog programs. Another direction is to consider smarter algorithms for local Datalog evaluation than semi-naive, by, for example, allowing to express unique-decomposition conditions (c.f., [5]) in the economic policy.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

2 Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Map-reduce extensions and recursive queries. In Anastasia Ailamaki, Sihem Amer-Yahia, Jignesh M. Patel, Tore Risch, Pierre Senellart, and Julia Stoyanovich, editors, *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 1–8. ACM, 2011. doi:10.1145/1951365.1951367.

3 Foto N. Afrati and Christos H. Papadimitriou. The parallel complexity of simple chain queries. In Moshe Y. Vardi, editor, *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23-25, 1987, San Diego, California, USA*, pages 210–213. ACM, 1987. doi:10.1145/28659.28682.

4 Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors, *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*, pages 99–110. ACM, 2010. doi:10.1145/1739041.1739056.

5 Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari, editors, *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 132–143. ACM, 2012. doi:10.1145/2247596.2247613.

6 Tom J. Ameloot, Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Parallel-correctness and transferability for conjunctive queries. In Tova Milo and Diego Calvanese, editors, *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 47–58. ACM, 2015. doi:10.1145/2745754.2745759.

7 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In Richard Hull and Wenfei Fan, editors, *Proceedings of the 32nd*

ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013, pages 273–284. ACM, 2013. doi:10.1145/2463664.2465224.

8 Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In Richard Hull and Martin Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 212–223. ACM, 2014. doi:10.1145/2594538.2594558.

9 Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 63–78. ACM, 2015. doi:10.1145/2723372.2750545.

10 Stavros S. Cosmadakis and Paris C. Kanellakis. Parallel evaluation of recursive rule queries. In Avi Silberschatz, editor, *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*, pages 280–293. ACM, 1986. doi:10.1145/6012.15421.

11 Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150, 2004. URL: http://www.usenix.org/events/osdi04/tech/dean.html.

12 Hasanat M. Dewan, Salvatore J. Stolfo, Mauricio A. Hernández, and Jae-Jun Hwang. Predictive dynamic load balancing of parallel and distributed rule and query processing. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994.*, pages 277–288. ACM Press, 1994. doi:10.1145/191839.191893.

13 Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. A framework for the parallel processing of datalog queries. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, pages 143–152. ACM Press, 1990. doi:10.1145/93597.98724.

14 Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. Parallel bottom-up processing of datalog queries. *J. Log. Program.*, 14(1&2):101–126, 1992. doi:10.1016/0743-1066(92)90048-8.

15 Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Parallel-correctness and containment for conjunctive queries with union and negation. *CoRR*, abs/1512.06246, 2015. arXiv:1512.06246.

16 Hadoop. http://hadoop.apache.org/.

17 Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the myria big data management service. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 881–884. ACM, 2014. doi:10.1145/2588555.2594530.

18 Paris C. Kanellakis. Logic programming and parallel complexity. In Giorgio Ausiello and Paolo Atzeni, editors, *ICDT'86, International Conference on Database Theory, Rome, Italy, September 8-10, 1986, Proceedings*, volume 243 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 1986. doi:10.1007/3-540-17187-8_27.

19 Bas Ketsman and Dan Suciu. A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 417–428. ACM, 2017. doi:10.1145/3034786.3034788.

20 Paraschos Koutris, Paul Beame, and Dan Suciu. Worst-case optimal algorithms for parallel query processing. In Wim Martens and Thomas Zeume, editors, *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016*, volume 48 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ICDT.2016.8.

21 Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In Maurizio Lenzerini and Thomas Schwentick, editors, *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 223–234. ACM, 2011. doi:10.1145/1989284.1989310.

22 Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 129–137. AAAI Press, 2014. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8505.

23 Jürgen Seib and Georg Lausen. Parallelizing datalog programs by generalized pivoting. In Daniel J. Rosenkrantz, editor, *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 29-31, 1991, Denver, Colorado, USA*, pages 241–251. ACM Press, 1991. doi:10.1145/113413.113435.

24 Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013. URL: http://www.vldb.org/pvldb/vol6/p1906-seo.pdf.

25 Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In Pablo Barceló and Reinhard Pichler, editors, *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*, volume 7494 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2012. doi:10.1007/978-3-642-32925-8_17.

26 Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1135–1149. ACM, 2016. doi:10.1145/2882903.2915229.

27 Apache spark. http://spark.apache.org/.

28 Jeffrey D. Ullman and Allen Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988. doi:10.1007/BF01762108.

29 Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015. URL: http://www.vldb.org/pvldb/vol8/p1542-wang.pdf.

30 Ouri Wolfson. Sharing the load of logic-program evaluation. In Sushil Jajodia, Won Kim, and Abraham Silberschatz, editors, *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, USA, December 5-7, 1988*, pages 46–55. IEEE Computer Society, 1988. doi:10.1109/DPDS.1988.675001.

31 Ouri Wolfson and Aya Ozeri. A new paradigm for parallel and distributed rule-processing. *SIGMOD Rec.*, 19(2):133–142, 1990. doi:10.1145/93605.98723.

32 Ouri Wolfson and Avi Silberschatz. Distributed processing of logic programs. *SIGMOD Rec.*, 17(3):329–336, 1988. doi:10.1145/971701.50242.

33 Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 13–24. ACM, 2013. doi:10.1145/2463676.2465288.

34 Weining Zhang, Ke Wang, and Siu-Cheung Chau. Data partition and parallel evaluation of datalog programs. *IEEE Trans. Knowl. Data Eng.*, 7(1):163–176, 1995. doi:10.1109/69.368511.

A Appendix

A.1 Proof for Theorem 5.2

▶ **Theorem 5.2.** PC$(Datalog, \mathcal{E}_{indep})$ *is undecidable.*

Proof. The proof is by reduction from the Datalog containment problem, which is well-known to be undecidable [1]. Let P_1 and P_2 be two arbitrary Datalog programs given as input for the containment problem. As usual, we assume that both are over the same output predicate, say O.

We first denote by P_i^* an indexed version of program P_i; particularly we define P_i^* as P_i in which all IDB predicates are annotated with index i. We now construct a program P by taking all rules from P_1^* and P_2^*, and adding the rules $O(\boldsymbol{x}) \leftarrow O_i(\boldsymbol{x})$, for $i \in \{1, 2\}$. We note that $edb(P) = edb(P_1^*) \cup edb(P_2^*)$ and $out(P) = \{O\}$. As economic policy we take $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ over the 2-node network $\{1, 2\}$. The consumption policy maps all facts with index i to server i. The production policy maps all facts with index i to server i, and all O-facts to server 2. The EDB facts are consumed on all servers.

Intuitively, programs P_1^* and P_2^* are computed locally on server 1 and server 2. It thus follows from the construction that (†) $P_1(I) \cup P_2(I) \subseteq [P, \boldsymbol{E}](I)$, for every instance I. Notice that rule $O(\boldsymbol{x}) \leftarrow O_1(\boldsymbol{x})$ is never used, since server 2 cannot consume facts over predicates with index 1.

It remains to show that $\boldsymbol{E}$ is parallel-correct for P if and only if $P_1 \subseteq P_2$. Indeed, if $P_1 \subseteq P_2$, then $O(I) = P_2(I)$ for every instance I, which implies that the policy will compute the correct result for O. The other direction follows from monotonicity of P. From (†) it follows that this condition is satisfied if and only if all facts over the O relation produced by $P(I)$ are also produced by $[P, \boldsymbol{E}](I)$, which is the case only if every fact $O(\boldsymbol{a}) \in P(I)$ implies a fact $O_2(\boldsymbol{a}) \in P(I)$. The latter is equivalent to saying $O(\boldsymbol{a}) \in P_1(I)$ implies $O(\boldsymbol{a}) \in P_2(I)$ for every instance I, which means that $P_1 \subseteq P_2$. ◀

A.2 Proof for Proposition 5.5

We first show the following Lemma.

▶ **Lemma A.1.** *For every proof tree T of depth d, there exists a proof tree $T' \sqsubseteq T$ of depth at most d that uses only minimal and useful rule instantiations.*

Proof. The proof is by induction on the depth of T, which we denote d. We show that there exists a proof tree $T' \sqsubseteq T$ with depth $\leq d$ that uses only minimal rule instantiations.

For the base case, let $d = 1$. Then, T corresponds to a single rule instantiation (τ, v) for P where all the facts in $v(body_\tau)$ are EDB facts. By definition, there is also a minimal rule instantiation (τ', v'), with $v'(head_{\tau'}) = v(head_\tau)$ and $v'(head_{\tau'}) \subseteq v(body_\tau)$, which admits the desired proof tree.

As induction hypothesis we take the statement of the lemma. Now for the induction step, suppose T has depth $d > 1$. Then, the root of T, together with its children, defines a rule instantiation (τ, v) for P. Now take an entailed minimal instantiation (τ', v) such that $v'(head_{\tau'}) = v(head_\tau)$ and $v'(body_{\tau'}) \subseteq v(body_\tau)$. For every fact $\boldsymbol{f} \in v'(head_{\tau'})$, let $T_{\boldsymbol{f}}$ be the subtree of T with root $\boldsymbol{f}$ (child of $root_T$). By the induction hypothesis, there is a proof

tree $T'_{\boldsymbol{f}} \sqsubseteq T_{\boldsymbol{f}}$ with depth $\leq d-1$ that uses only minimal rule instantiations. The proof tree that combines instantiation (τ', v') with $T'_{\boldsymbol{f}}$ for all $\boldsymbol{f} \in v'(\tau')$ is as desired. ◀

▶ **Proposition 5.5.** *For every Datalog program P, we have $\mathcal{N}_P^{ess} \subseteq \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$.*

Proof. The containment $\mathcal{N}_P^{ess} \subseteq \mathcal{N}_P^{use}$ is straightforward, since a proof tree does not use any useless rule instantiations. We next show that $\mathcal{N}_P^{ess} \subseteq \mathcal{N}_P^{min}$. Suppose that we have an instantiation of rule τ with valuation v that is essential. Then, there exists some fact $\boldsymbol{f}$ and instance I for which every proof tree T has a vertex $\boldsymbol{g}$ with $\boldsymbol{g} = v(\mathit{head}_\tau)$ and $v(\mathit{body}_\tau) \subseteq \mathit{children}_T(\boldsymbol{g})$. By Lemma A.1, we can pick this tree such that it uses only minimal rule instantiations. This implies that the rule instantiation with head $\boldsymbol{g}$ and body $\mathit{children}_T(\boldsymbol{g})$ is minimal. Hence, the instantiation with head $v(\mathit{head}_\tau)$ and body $v(\mathit{body}_\tau)$ is also minimal. ◀

A.3 Proof for Proposition 5.7

We first show the following lemma.

▶ **Lemma A.2.** *Let P be a Datalog program and $\boldsymbol{E}$ an economic policy. If a proof tree T for P is supported by $\boldsymbol{E}$, then for every instance I, with $\mathit{fringe}_T \subseteq I$, we have $\mathit{root}_T \in [P, \boldsymbol{E}]$.*

Proof. The proof is by induction on the depth d of T. Particularly we show using a simple inductive argument that $\mathit{root}_T \in \mathsf{local}_i^k$, for some server i and $k \leq d$, which implies $\mathit{root}_T \in [P, \boldsymbol{E}]$. Recall that local_i^k denotes the facts residing locally on server i after the k-th computation round.

As base case let $d = 1$, meaning that T describes a single rule instantiation. After the first communication round, all servers j have $\mathsf{local}_j^0 \cup \mathsf{rec}_j^1 \subseteq I \cap \mathsf{facts}_{\boldsymbol{C}}(j)$. By the assumption that $\boldsymbol{E}$ supports T, it follows that $\mathit{children}_T(\mathit{root}_T) \subseteq \mathsf{facts}_{\boldsymbol{C}}(i) \cap I \subseteq \mathsf{local}_i^1$ and $\mathit{root}_T \in \mathsf{facts}_{\boldsymbol{P}}(i)$, for some server i, thus after the first computation round, $\mathit{root}_T \in \mathsf{local}_i^1$.

For $d > 1$ we observe that root_T and its children in T define a rule instantiation (τ, v), and, by the assumptions of the lemma, this rule instantiation is supported by $\boldsymbol{E}$. More specifically, some server i exists where $\mathit{root}_T \in \mathsf{facts}_{\boldsymbol{P}}(i)$ and $\mathit{children}_T(\mathit{root}_T) \subseteq \mathsf{facts}_{\boldsymbol{C}}(i)$. Further, for all facts $\boldsymbol{f} \in \mathit{children}_T(\mathit{root}_T)$, the respective subtree $T_{\boldsymbol{f}}$ of T with root $\boldsymbol{f}$ is supported by $\boldsymbol{E}$ and with depth $d-1$. By the induction hypothesis it follows that for all these facts $\boldsymbol{f}$ there is a server j and $k \leq d-1$, where $\boldsymbol{f} \in \mathsf{local}_j^k$. Therefore $\mathit{children}_T(\mathit{root}_T) \subseteq \mathsf{local}_i^{k^*} \cup \mathsf{rec}_i^{k^*}$, where k^* denotes the maximal k, and consequently, $\mathit{root}_T \in \mathsf{local}_i^{k^*+1} \subseteq \mathsf{local}_i^d$. ◀

We say that an economic policy $\boldsymbol{E}$ *supports a proof tree* T if all the rule instantiations in T are supported.

▶ **Lemma A.3.** *Let P be a Datalog program. An economic policy $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C}; U)$ is parallel-correct for P if and only if for every proof tree for P with fringe over $\mathsf{facts}(\sigma(P), U)$, an entailed supported proof tree exists.*

Proof. *(If).* Let I be an arbitrary instance, we show $P(I) = [P, \boldsymbol{E}](I)$. By monotonicity, $[P, \boldsymbol{E}](I) \subseteq P(I)$, thus we focus on completeness. For this, let $\boldsymbol{f} \in P(I)$, which means that a proof tree T exists with $\mathit{fringe}_T \subseteq I$ and $\mathit{root}_T = \boldsymbol{f}$. Particularly, by the assumption of the lemma we can choose T so that it is also supported by $\boldsymbol{E}$. It now follows from Lemma A.2 that $\boldsymbol{f} \in [P, \boldsymbol{E}]$.

(Only if). We assume $(\boldsymbol{P}, \boldsymbol{C})$ is parallel-correct for P. Let T be an arbitrary proof tree. The proof is by construction following the derivation of root_T using $\boldsymbol{E}$. First, from parallel-correctness it follows that $P(I) = [P, \boldsymbol{E}](I)$, for any instance I. Here we take $I = \mathit{fringe}_T$,

implying $root_T \in [P, \boldsymbol{E}](I)$. The proof now continues by induction on the number of rounds needed for $\boldsymbol{E}$ to derive $root_T$.

The induction hypothesis is that if k rounds are needed to derive $root_T$, then a supported proof-tree of depth k entailed by T exists.

As a base case suppose $k = 1$. That is, $root_T \in \mathsf{local}_i^1$, meaning that $root_T \in P_{\uparrow \boldsymbol{E}}(\mathsf{local}_j^0 \cup \bigcup_j \mathsf{rec}_j^1)$ for some server j. Particularly, a valuation v and rule $\tau \in P$ existed with $v(body_\tau) \subseteq \mathsf{facts}_{\boldsymbol{C}}(j) \cap I$ and $v(head_\tau) = root_T$, which means that the corresponding rule instantiation is supported by $\boldsymbol{E}$. Here, the proof tree admitted by (τ, v) is as desired.

For $k > 1$ the proof is analogous, but now we take as proof tree the tree obtained by concatenating the rule instantiation with the proof trees for each child. Existence of the latter follows from the induction hypothesis. As the number of rounds decreases by one in each inductive step, and the fringes of the obtained trees cannot have other facts than does in I, the constructed proof tree is as again as desired. ◀

▶ **Proposition 5.7.** *Let P be a Datalog program and $\boldsymbol{E}$ an economic policy. If $\boldsymbol{E}$ supports all minimal and useful rule instantiations in P, then it is parallel-correct. If $\boldsymbol{E}$ is parallel-correct for P, then it supports all essential rule instantiations.*

Proof. The first item follows from Lemma A.3 and Lemma A.1. For the second item, consider a parallel-correct policy $\boldsymbol{E}$ and an essential instantiation of rule τ with valuation v. By the definition of essential, for some fact $\boldsymbol{f}$ and instance I, every proof tree T for $\boldsymbol{f}$ on I and P has a vertex $\boldsymbol{g}$ with $\boldsymbol{g} = v(head_\tau)$ and $v(body_\tau) \subseteq children_T(\boldsymbol{g})$. By Lemma A.3, there must exist such a tree T that is supported. This implies that there exists server s with $v(head_\tau) = \boldsymbol{g} \in \mathsf{facts}_{\boldsymbol{P}}(s)$ and $v(body_\tau) \subseteq children_T(\boldsymbol{g}) \subseteq \mathsf{facts}_{\boldsymbol{C}}(s)$. Hence, the essential rule instantiation is indeed supported. ◀

A.4 Proof for Proposition 5.8

▶ **Proposition 5.8.** *Let P be a Datalog program where each* IDB *predicate occurs only in the head of rules (i.e., P is a union of CQs). Then, $\mathcal{N}_P^{ess} = \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$.*

Proof. Because P is not recursive, $\mathcal{N}_P^{use} = \mathcal{N}_P^{all}$; hence, because of Proposition 5.5 it suffices to show that $\mathcal{N}_P^{min} \subseteq \mathcal{N}_P^{ess}$. Indeed, consider a minimal instantiation for rule τ with valuation v, and consider the instance $I = v(body_\tau)$ and fact $\boldsymbol{f} = v(head_\tau)$. Take any proof tree T for $\boldsymbol{f}$ on I and P; T must have depth one. Because of the minimality of the rule instantiation, it must be that $children_T(\boldsymbol{f}) = v(body_\tau)$, which proves the essentiality. ◀

A.5 Proof for Proposition 6.2

▶ **Proposition 6.2.** *Let P be a Datalog program. Every GHP $\boldsymbol{E}$ for P is strongly supporting for P and, as a consequence, parallel-correct for P.*

Proof. To show that $\boldsymbol{E}$ is supporting, consider some rule $\tau \in P$, and its instantiation w.r.t. some valuation v. Consider some atom $A = R(\boldsymbol{y})$ in the body of τ; then the consumption policy says that its instantiation $\boldsymbol{f} = v(A)$ will be consumed in the set $S^\tau_{\boldsymbol{f},A}$, as defined in Section 6. Similarly if A is the head, the fact $\boldsymbol{f}$ will be produced in $S^\tau_{\boldsymbol{f},A}$. Now we can write the intersection $\bigcap_{A \in \tau} S^\tau_{\boldsymbol{f},A}$ as:

$$\bigcap_{A \in \tau} \{\mathsf{map}^{\chi(\tau)}(\mathbf{q}) \mid \forall i : \emptyset \subsetneq \rho^\tau(i) \subseteq vars(A) \Rightarrow \mathbf{q}_i = h_i^{\chi(\tau)}(v(\rho^\tau(i)))\}$$

$$\supseteq \{\mathsf{map}^{\chi(\tau)}(\mathbf{q}) \mid \forall i : \mathbf{q}_i = h_i^{\chi(\tau)}(v(\rho^\tau(i)))\} \supsetneq \emptyset$$

In other words, there will be at least one machine in $\bigcap_{A \in \tau} S^{\tau}_{A}$, which means that every instantiation of the rule τ will be strongly supported. ◀

A.6 Example A.4

▶ **Example A.4.** For an example showing that not every 1-bounded GHP is weakly pivoting, consider the following *non-pure* Datalog program P:

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

and GHP family $\mathcal{H}$ over a single one-dimensional cube 1. Let map^1 be the identity mapping, $\chi(\tau) = 1$ and $\rho^{\tau}(1) = \{x\}$ for all rules τ. Clearly, $\mathcal{H}$ is not weakly pivoting. Nevertheless, it can be shown that $\mathcal{H}$ is 1-bounded, which follows from the observation that only single-valued rule instantiations can satisfy under P.

A.7 Example A.5

▶ **Example A.5.** Consider the left-linear TC example, and let $\sigma = \{T\}$. Suppose we choose $\boldsymbol{\beta}(T) = (1)$. Then $\boldsymbol{\beta}$ is a pivot base for σ, since for the recursive rule and the only pair of T-atoms $T(x,y), T(x,z)$ we have $T(x,y)[\boldsymbol{\beta}(T)] = T(x,y)[1] = (x)$, and $T(x,z)[\boldsymbol{\beta}(T)] = T(x,z)[1] = (x)$. Since T is the only IDB relation, left-linear TC is pivoting.

Next, consider the left-linear TC with an extra rule:

$$T(x,y) \leftarrow R(x,y). \qquad T(x,y) \leftarrow T(x,z), R(z,y). \qquad U(y) \leftarrow T(x,y).$$

Here, there are two IDB relations, but only T occurs in the body of a rule. The pivot base $\boldsymbol{\beta}$ from before is still a pivot base for $\{T\}$; hence the program is weakly pivoting. However, there is no pivot base for to $\{T, U\}$, which means that the program is not pivoting.