On the CALM Principle for Bulk Synchronous Parallel Computation

Matteo Interlandi
DIEF, Università di Modena e Reggio Emilia
matteo.interlandi@unimore.it

Letizia Tanca
DEIB, Politecnico di Milano
letizia.tanca@polimi.it

ABSTRACT

In the recent years a lot of emphasis has been placed on two apparently disjoined fields: *data-parallel* [16, 26, 11, 32] and *eventually consistent distributed systems* [30, 12, 17]. In this paper we propose a theoretical study over an *eventually consistent data-parallel computational model*. The keystone is provided by the recent finding that a class of programs exists which can be computed in an eventually consistent, coordination-free way: *monotonic programs* [21]. This principle is called *CALM* and has been proven for distributed asynchronous settings [6]. We make the case that, using the techniques developed by Ameloot et al., CALM does not hold in general for data-parallel systems, wherein computation usually proceeds synchronously in rounds and where communication is reliable. We then show that using novel techniques subsuming the one of [6], the satisfiability of the CALM principle is directly related with the assumptions imposed on the behavior of the system.

1. INTRODUCTION

Recently, a lot of emphasis has been placed on how distributed systems can exploit different levels of *consistency* in order to improve performances on specific tasks and network configurations, while maintaining correctness [30, 12, 17]. Another topic strictly related to consistency is *coordination*, usually informally interpreted as a mechanism which permits to achieve a distributed agreement on some property of a system [18]. If we consider consistency as such, coordination can be seen as a tool able to enforce consistency when, in the natural execution of a system, this property is not achievable in general. For instance, in MapReduce, reduce tasks start their computation strictly after map tasks have completed (coordination). This permits to avoid situations in which late-arriving values can jeopardize the final outcome (consistency).

Recent research has identified a class of programs that can always be computed in an eventually consistent, coordination-free way: *monotonic programs* [21]. This principle is called *CALM* (Consistency and Logical Monotonicity) and has been proven in [6] in the case of asynchronous systems. In this paper we study the theoretical results springing from the application of the CALM principle over *synchronous systems with reliable communication* (*rsync*). It is in fact well-known that *rsync* is a common setting in modern data-parallel frameworks such as MapReduce and Pregel. In this type of systems, computation is commonly performed in *rounds*, where each task is blocked and cannot start the new round untill a *synchronization barrier* is reached, i.e., before every other task has completed its local computation. Yet, by extending the CALM principle over *rsync* systems, we would be able to formally distinguish the cases where a synchronous "blocking" computation is actually required by the program semantics – and therefore must be strictly enforced by the system – from those where, instead, an asynchronous pipelined execution can be performed underneath as optimization. It is is fact best practice, in database systems, to prefer pipelining over the full materialization of partial results, when possible. Far batch parallel processing, the benefits of understanding where the former can be substituted over the latter are considerable [14]: thanks to the fact that data is processed as soon as it is produced, (i) *online computation* is possible, i.e., the final result can be estimated and refined during the execution. This brings to (ii) *decrease response time*, (iii) *better resource utilization*, and (iv) new data can be incrementally added to the input making possible *continuos computation*. Over all pipelining is highly desirable in "Big Data" contexts where full materialization is often problematic, e.g., it is well-known that the scalability of Spark [32] is limited by the available main memory.

Surprisingly enough, however, the demonstration of the CALM principle in *rsync* systems is not trivial. As matter of fact, with the communication model and the notion of coordination as defined in [6], the CALM principle does not hold in general in *rsync* settings. Thus, in order to extend the CALM principle over data-parallel synchronous computation, in this paper we will develop a new *generic parallel computation model* leveraging previous works on *relational transducers* [2] and *transducer networks* [6], and grounding *rsync* computation on the well known *Bulk Synchronous Parallel* (BSP) model [29]. In fact, nowadays data-parallel frameworks are for the majority not only assuming the underlying system to be *rsync*, but they are also directly [26] or indirectly [28] implementing the BSP model. Exploiting this new type of transducer network, we will then show that the CALM principle is satisfied for synchronous and reliable systems under a new specific definition of coordination-freeness but, surprisingly enough, only for a subclass of monotonic queries. When defining coordination-freeness we will take advantage of recent results describing how knowledge can be acquired in synchronous systems [9, 10].

The contributions of the paper are thus as follows:

- Using the techniques developed in [6], the CALM principle is proven to not hold in general for *rsync* systems;
- A novel computational model is introduced which emulates common patterns found in modern data-parallel frameworks;

- A new definition of coordination is proposed and the satisfiability of the CALM principle is shown to be directly linked to the assumptions imposed on the behavior of the system;
- A complete taxonomy of queries is provided which permits to identify different types of coordination patterns; and
- The new definition of coordination is shown to subsume the previous formalization used in [6].

The rest of the paper is organized as follows: Section 2 introduces some preliminary notation. Section 3 defines our model of synchronous and reliable distributed parallel computation, and shows that the CALM principle is not satisfied for systems of this type. Section 4 proposes a new computational model based on hashing, while Section 5 introduces a new definition of coordination. Finally, Section 6 discusses CALM under the new setting. The paper ends with the discussion of related work and some concluding remarks.

2. PRELIMINARIES

Although we expect the reader to be familiar with the basic notions of database theory, we exploit this section to set forth our notation which is close to that of [1] and [7].

2.1 Basic Notations

We denote by $\mathcal{D}$ an arbitrary *database schema* composed by a non empty set of *relation* (*schemas*) R. In the following we will use the notation $R^{(a)}$, or equivalently (R, a), to denote a relation name together with its *arity* a. With **dom** we indicate a countably infinite set of *constants*. Given a relation $R^{(a)}$, a *fact* $R(\bar{\mathtt{u}})$ over R is an ordered a-tuple composed by constants only. A *database instance* is a finite set of facts $\mathbf{I}$ over the relations of $\mathcal{D}$, while a *relation instance* $I_R \subseteq \mathbf{I}$ is a set of facts defined over R, with $R \in \mathcal{D}$. The set $adom(\mathbf{I})$ of all constants appearing in a given database instance $\mathbf{I}$ is called *active domain* of $\mathbf{I}$, while $inst(\mathcal{D})$ denotes the set of all the database instances defined over $\mathcal{D}$. Given a database schema $\mathcal{D}$, a subset of $\mathcal{D}$ called $\mathcal{D}_{in}$, and a relation $R \in \mathcal{D}$, a *query* q_R is a partial function such that $q_R : inst(\mathcal{D}_{in}) \to inst(R)$ and $adom(q(\mathbf{I})) \subseteq adom(\mathbf{I}) \subseteq$ **dom**. In practice we will only consider *generic* queries, i.e., if p is a permutation of **dom**, and $\mathbf{I}$ an input instance, then $q(p(\mathbf{I})) = p(q(\mathbf{I}))$. Note that in our definition we have considered queries with a single output relation. This is not a limitation since queries with multiple output relations can be expressed as a union thereof. In such case, given an input and output schema $\mathcal{D}_{in}$ and $\mathcal{D}_{out}$, we will write $\mathcal{Q} = \{q_R | R \in \mathcal{D}_{out}\}$. We will sometime employ the function sch to return from a query its schema, i.e., $\mathcal{D} = sch(\mathcal{Q})$. In this paper we will consider the following query languages, all expressible using well-known rule-based formalisms: union of conjunctive queries *UCQ*, first order logic *FO*, *Datalog*, and Datalog with negation *Datalog*$^\neg$. For this latter language we will assume the *stratified semantics*.

We define a *distributed system* to be a *fully connected* graph of communicating nodes $N = \{1, \ldots, n\}$. To each node i we assign a *node configuration* denoted by the pair (N, i). We will in general assume each node to share the same *global* database schema. We will use the notation I_R^i to denote a *local instance* for node i over R, while a *global instance* over R is defined as $I_R = \bigcup_{i \in N} I_R^i$. Given a database initial instance $\mathbf{I}$ defined over a subset of the global schema, a *partition function* P exists which maps each node to a portion of the intial instance $P(\mathbf{I}, i) = \mathbf{I}^i$. Finally, a *network configuration* is identified with the pair (N, P).

2.2 Relational Transducers

In order to be able to define, in Section 3, how generic computations can be performed in synchronous distributed settings, here we introduce a special version of *relational transducer* [2] named *timed relational transducer* (hereafter simply transducer). Informally, a transducer is a transition system programmed as a deductive database. We consider each transducer as composed by an immutable *database*, a *memory* used to maintain data among consecutive system transitions, an internal *time*, and a *system* configuration. In addition, a transducer can produce an *output* for the user and can also *emit* some data towards other transducers (data emission will be clarified in Section 3 with the concept of *transducer network*). More formally, a transducer $\mathcal{T}$ is defined by the pair $(\mathcal{P}, \Upsilon)$ where $\mathcal{P}$ and Υ respectively denote the *transducer program* and *transducer schema*. A transducer schema Υ is represented by a 6-tuple (Υ_{db}, Υ_{mem}, Υ_{emt}, Υ_{out}, Υ_{time}, Υ_{sys}) of disjoint relational schemas, respectively called $database$, $memory$, $emit$, $output$, $time$ and $system$ schemas. As default, we consider Υ_{sys} to contain two unary relations `Id`, `All`, while *time* includes just the unary relation `Time`, which is employed to store the transducer current local *clock* value. A *transducer local state* over the schema Υ is then an instance I over $\Upsilon_{db} \cup \Upsilon_{mem} \cup \Upsilon_{out} \cup \Upsilon_{sys}$. The transducer program $\mathcal{P} = Q_{ins} \bigcup Q_{del} \bigcup Q_{out} \bigcup Q_{emt}$ is composed by a collection of *insertion*, *deletion*, *output* and *emission* queries all taking as input an instance over the schema Υ, where $Q_{ins} = \{q_R^{ins} | R \in \Upsilon_{mem}\}$, $Q_{del} = \{q_R^{del} | R \in \Upsilon_{mem}\}$, $Q_{out} = \{q_R^{out} | R \in \Upsilon_{out}\}$, and $Q_{emt} = \{q_R^{emt} | R \in \Upsilon_{emt}\}$.

Starting from a relational transducer $\mathcal{T} = (\mathcal{P}, \Upsilon)$ and a node configuration (N, i), we can construct a *configured transducer*, denoted by $\mathcal{T}_N^i$ by setting $I_{Id} = \{\mathtt{Id(i)}\}$ and $I_{All} = \{\mathtt{All(j)} | \mathtt{j} \in N\}$. Given a configured transducer and an initial instance $\mathbf{I}$ defined over Υ_{db}, we create a *transducer initial local state* by setting $I_{db} = \mathbf{I}$. Now, let I_{in}, J_{emt} be two instances over Υ_{emt}, with the former identifying a set of previously emitted facts. If I is a local state, a *transducer transition*, denoted by $I, I_{in} \Rightarrow J, J_{emt}$ is such that J is the updated local state, while J_{emt} contains a set of emitted facts that must be addressed to other transducers. The semantics for updates leaves the $database$ and the $system$ instances unchanged, while in each $memory$ relation the facts produced by the insertion query Q_{ins} are inserted, and all the facts returned by the deletion query Q_{del} removed. In case of conflicts, i.e., a fact is simultaneously added and removed, we adopt the no-op semantics. As a result for the user, the set of tuples resulting from the Q_{out} query are output. For what concerns J_{emt}, this is the set of facts returned by query $Q_{emt}(I \cup I_{in})$ and emitted by the transducer towards the other nodes. We assume, that once emitted, the facts cannot be retracted. A transducer transition $I, I_{in} \Rightarrow J, J_{emt}$ is hence formally defined by the following transition laws:

- J and I agree on Υ_{db} and Υ_{sys};
- $J_{mem} = (I_{mem} \cup I_{ins}^+) \setminus I_{del}^-$, where $I_{ins}^+ = Q_{ins}(I \cup I_{in}) \setminus Q_{del}(I \cup I_{in})$ and $I_{del}^- = Q_{del}(I \cup I_{in}) \setminus Q_{ins}(I \cup I_{in})$;
- $J_{out} = I_{out} \cup Q_{out}(I \cup I_{in})$; and
- $J_{emt} = Q_{emt}(I \cup I_{in})$.

Indeed transitions are deterministic, i.e., if $I, I_{in} \Rightarrow J$, J_{emt} and $I, I_{in} \Rightarrow J', J'_{emt}$, then $J = J'$ and $J_{emt} = J'_{emt}$.

Many different versions of transducers can be obtained by constraining the type of queries or the transducer schema. A transducer is *space oblivious* if its queries do not use any $system$ relation. Intuitively, this means that each query is unaware of the configuration, because it is independent both of the node it is running on,

and of the other nodes composing the network. Instead, a transducer is *time oblivious* if its queries do not use the relation `Time`. This means that each query is unaware of the time point in which the computation is. In general a transducer is *oblivious* if it is both time and space oblivious. A transducer is called *monotone* if all of its queries are monotone. Finally, we say that a transducer is *inflationary* if $memory$ facts are never deleted – i.e., Q_{del} is empty.

Remark: As defined the relational transducer model is general. However it can be instantiated using a specific query language $\mathcal{L}$. We shall then write $\mathcal{L}$-transducer to denote that the queries are actually implemented in $\mathcal{L}$.

In the following examples, just for the sake of readability, we will use subscripts in the rule heads to indicate the transducer query the corresponding relation belongs to.

Example 1. As a first example of relational transducer, let us compute a join between two relations $R^{(2)}$ and $T^{(2)}$. This can be accomplished by a UCQ-transducer $\mathcal{T}$ as follows:

Schema: $\Upsilon_{db} = \{R^{(2)}, T^{(2)}\}, \Upsilon_{out} = \{Q^{(3)}\}$
Program: $Q_{out}(u, v, w) \leftarrow R(u, v), T(v, w).$

Let now $\mathcal{T}_N^i$ be a configured version of $\mathcal{T}$, and $\mathbf{I}$ an initial instance defined over Υ_{db}. We then have that a transition for $\mathcal{T}_N^i$ is such that $I, I_{in} \Rightarrow J, J_{emt}$, where $I = \mathbf{I} \cup I_{sys}$, I_{in} and J_{emt} are empty (no emission query exists), and $J = \mathbf{I} \cup I_{out} \cup I_{sys}$, where I_{out} is the result of the query over Q, i.e., the join of R and T.

3. COMPUTATION IN SYNCHRONOUS AND RELIABLE DISTRIBUTED SYSTEMS

Query computability is usually defined using the classical model of computation, i.e. the Turing machine. However in this paper we are interested in distributed computation. But what is the meaning of computing a query in distributed settings? In order to answer this question, in the following we will introduce a novel *transducer network* [6] where computation is *synchronous*, and communication is *reliable*. Such kind of transducer network permits us to define how a set of relational transducers can be assembled in order to obtain a computational model for distributed data-parallel systems.

3.1 Transducer Networks

A *timed homogeneous transducer network* (henceforth *transducer network* or *specification*) $\mathcal{N}$ is a tuple $(\mathcal{T}, \mathcal{T}^e, \sigma, \gamma)$ where $\mathcal{T}$ is a transducer, $\mathcal{T}^e$ is a special transducer defining the *environment*, σ is a *state* function mapping each node i to a local state I^i, while γ is a *communication* function mapping each node to a set of input facts I_{in}^i. Being the network homogeneous, all nodes employ the same transducer $\mathcal{T}$, while the only thing that we allow to change from node to node is its state. The transducer network abstraction is thus appropriate for modeling data-parallel computation, where each node applies the same set of transformations in parallel over a partition of the initial data given as input.

For the moment we will consider two types of networks: *broadcasting* and *communication-free*. The former are the networks in which the function γ is such that every fact emitted by a transducer is sent to all the other transducers composing the network, i.e., $I_{in}^i = \bigcup_{j \in N} J_{emt}^j$. The latter instead is such that every fact is delivered just locally to the emitting node, i.e., $I_{in}^i = J_{emt}^i$. We dubbed these networks as communication-free because, intuitively, no tuple is actually communicated to the other nodes. We will use the labels b, f when we want to point out that a particular transducer network is respectively broadcasting or communication-free.

We assume the environment to be a "special" relational transducer collecting all the facts emitted by the transducers composing the network. More precisely, we define $\mathcal{T}^e = (\mathcal{P}^e, \Upsilon^e)$ as a transducer where Υ^e is composed only by $memory$ and $emit$ relation schemas ($database$, $output$ and $time$ schemas are empty, while $system$ is as usual), where $\Upsilon_{emt}^e = \Upsilon_{emt}$, and Υ_{mem}^e is the primed version of Υ_{emt}^e. For what concern the transducer program, for each $R \in \Upsilon_{emt}$, $\mathcal{P}^e$ is composed by a set of queries in the form:

$$R'_{ins}(\bar{u}) \leftarrow R(\bar{u}).$$
$$R_{emt}(\bar{u}) \leftarrow R(\bar{u}).$$

used to memorize the received tuples, and to emits every received fact. Given a network configuration (N, P) we denote with $\mathcal{N}_{N,P}$ a *configured transducer network*, i.e., a specification where all the transducers have being configured, and where each node i will receive a database following the partition function P. When $|N| = 1$, we always assume P to return the full instance, i.e., $P(\mathbf{I}, i) = \mathbf{I}$. We call this the *trivial configuration*. A *transducer network global state* is now a tuple $(I^e, I^1, \ldots, I^n)$ where for each $j \in N \cup e$ the j-th element is the related relational transducer state I^j. The definitions introduced at the end of Section 2.2 can be naturally generalized over transducer networks.

3.2 Network Run

At any point in time each transducer is in some particular local state incapsulating all the information of interest the node possesses. In distributed settings, however, we deal with multiple transducers, therefore local states are not enough to represent the status of the system. This is because transducers interact with each other, and hence a proper machinery is needed in order to consider the system as a whole. Therefore, given a transducer specification $\mathcal{N}$, we define how transducer network global states may change over time through the notion of *run*, which binds logical time values to global states. If we assume time values to be isomorphic to the set of natural numbers, the function ρ is defined as a mapping between $\mathbb{N}_0$ and the set of all possible global states. If $\rho(s) = (I^e, I^1, ..., I^n)$ is the transducer network global state at time s, with $\rho^i(s)$ we denote the transducer state I^i for node $i \in N$, while the initial global state $\rho(0)$ is such that $I_{db}^i = P(\mathbf{I}, i)$, and the local $system$ relations are properly initialized, while I^e is empty (except the $system$ relations). We refer to the pair (ρ^i, s) consisting of a run ρ and a time s for node i as a *point*.

A distributed system may have many possible runs, indicating all the possible ways the global state can evolve. In order to capture this, starting from a configured transducer network $\mathcal{N}_{N,P}$ and an initial instance $\mathbf{I}$, we define a *system* $\mathcal{S}_{\mathcal{N}_{N,P},\mathbf{I}}$ as a set of runs, where $\mathcal{N}_{N,P}, \mathbf{I}$ denote the *system context*. Using this definition we are able to characterize a distributed system not just as a collection of interacting nodes, but directly by modeling its behavior. We will often employ the signature $\mathcal{S}$ to denote a system if its context is irrelevant or clear from the domain. In the following we will be also interested in defining *classes* of systems, i.e., set of systems having identical specification but different configurations. Thus, if a system is defined starting from a configured transducer network and an instance, a class of systems is defined starting from a simple specification $\mathcal{N}$ and by adding some constraint to its configuration. If we bind all the parameters, we obtain a specific system $\mathcal{S}_{\mathcal{N}_{N,P},\mathbf{I}}$.

3.3 Bulk Synchronous and Reliable Systems

As said previously, we are mainly interested in synchronous systems with reliable communication.

Definition 1. A *synchronous system* $\mathcal{S}^{sync}$ is a set of runs fulfilling the following conditions:

S1 A global clock is defined and accessible by every node;

S2 The relative difference between the time clock values of any two nodes is bounded; and

S3 Emitted tuples arrive at destination at most after a certain bounded physical time Δ.

The first property can be expressed in our framework by linking the time value stored for each node in the `Time` relation with the external time used to reason about system runs. This can be accomplished by defining a *timed local transition* $I, I_{in} \stackrel{s}{\Rightarrow} J, J_{emt}$ as a local transition where, at each time instant s, $I_{time} = \texttt{Time}(s)$. In this new setting, we then have that each transducer accepts as input also the clock value. The environment can be employed to directly provide the clock driving the computation for all the transducers. To accomplish this, we can add to Υ^e_{mem}, Υ^e_{emt} respectively, the new relations `Time`, and `STime` (where `STime` is for synchronous time), while $\mathcal{P}^e$ is augmented with the following clauses:

$$\texttt{Time}_{ins}(s) \leftarrow \texttt{Time}(t), \texttt{Succ}(t, s). \quad (1)$$

$$\texttt{Time}_{del}(t) \leftarrow \texttt{Time}(t). \quad (2)$$

$$\texttt{STime}_{emt}(t) \leftarrow \texttt{Time}(t). \quad (3)$$

where eq. (1) is used to move forward the current round, and eq. (3) emits the new clock value. $\mathcal{T}$ is finally equipped with the rule:

$$\texttt{Time}(t) \leftarrow \texttt{STime}(t). \quad (4)$$

Under this perspective, each timed local transition is basically labeled with the time value in which it is performed. The second property can be implemented by assuming that programs proceed in *rounds*, and that each round, operationally speaking, lasts enough to permit each node computation to reach the fixpoint[1]. In the following, w.l.o.g. we will use the round number to refer to the time clock. In order to express the third property, we assume that every emitted fact is first buffered locally, and then, once the node has completed its local transition, bulky delivered to destination. We can therefore infer that Δ is composed by two quantities: the network latency δ, and a *synchronization delay* which expresses the time required by the system to detected that every node has reached the fixpoint. By construction, the network latency is considered twice in the definition of Δ, once from the emitting node to the environment, and once from the environment to the receiving node. We call the new type of transducer network so obtained as *bulk synchronous* or simply *synchronous*. We use the tuple (N, t, P) to specify a configuration for a synchronous transducer, where N is the set of nodes, t is the initial round value and P a partition function. $\mathcal{N}_{N,t,P}$ then specifies a configured bulk synchronous network.

Consider now a function θ mapping a round number to the physical time in which it occurs. We simplify our model by adding to property **S3** the following constraint named *bounded variance* :

S3′ for every pair of consecutive rounds s, s', if $var = max(\delta) - min(\delta)$, then $var \ll \theta(s') - \Delta - \theta(s)$

Informally, condition **S3′** specifies that, between two consecutive rounds, the variance of the communication delay is amply lower than the time spent for computation. From the above assumptions it follows that each tuple derived by an emission query at round s, will be available at the receiving site not later than physical time $\theta(s+1) + var$ and that $\theta(s+1) \leq \theta(s+1) + var < \theta(s+2)$, i.e., tuples are received during the successive round. Henceforth we will use the signature $\mathcal{S}^{sbv}_{\mathcal{N}_{N,t,P},\mathbf{I}}$ to denote a synchronous system with bounded variance (*sbv*).

Although under the *sbv* semantics the variance is bounded, the actual instant in which a tuple is received falls non-deterministically in the range $[\theta(s+1), \theta(s+1) + var]$. To further simplify the model, we can add to **S3′** an extra condition named *fixed delivery*:

S3″ Every tuple emitted at round s is delivered exactly at time $\theta(s+1) + var$.

In this situation, we can safely shift the beginning of each round so that every tuple is precisely delivered at $\theta(s')$ when emitted at round s, where $s' = \theta^{-1}(\theta(s+1) + var)$. Naturally this is possible only because we are assuming a bounded variance. W.l.o.g., for simplicity of notation in the following we will write $s+1$ instead of s'. We will then use the signature $\mathcal{S}^{sfd}_{\mathcal{N}_{N,t,P},\mathbf{I}}$ to denote a synchronous system with fixed delivery (*sfd*).

Following the above properties we have that a remote tuple is either received after a bounded amount of time or never received. Initially, however, we have made the assumption on each system to be reliable, i.e., all emitted tuples arrive at destination.

Definition 2. A synchronous system to be reliable must satisfy properties **S1** - **S3** in addition to the following two conditions:

R1 In every run, for all s, s' s.t. $s < s'$, for every input fact for node i at round s', there exists a corresponding emission query that has been satisfied on node j in round s; and

R2 In every run, for all nodes i, j, and rounds s, s' with $s < s'$, if a fact has been emitted by node i at round s, then there exists a time s' such that the same fact belongs to the input instance state of node j at round s'.

Informally, properties **R1** - **R2** specify that if an emitted fact exists in an instance at a given round, then it has been generated by an emission query in a previous round, and, vice-versa, if an emission query derives a new fact, then that fact must appear in a successive round into the local state of a node. Finally, we denote with $\mathcal{S}^{rsync}$ the system satisfying conditions **S1** - **S3** and **R1** - **R2**, while a *reliable synchronous system with bounded variance* $\mathcal{S}^{rsbv}$ is a set of runs fulfilling **S3′** in addition to **S1** - **S3** and **R1** - **R2**, and with $\mathcal{S}^{rsfd}$ we denote a *reliable synchronous system with fixed delivery*: this is a $\mathcal{S}^{rsbv}$ where also condition **S3″** hold. This later system exactly simulates how real-world data-parallel frameworks behave. Except for Section 6, we will then only consider *rsfd* systems. In the next section we will hence show how all the required properties are enforced during global transitions.

3.4 Global Transitions

Given a transducer network $(\mathcal{T}, \mathcal{T}^e, \sigma, \gamma)$ and two global states $\mathbf{F} = (I^e, I^1, ..., I^n)$, $\mathbf{G} = (J^e, J^1, ..., J^n)$, a *global transition* for a *rsfd* system, denoted by $\mathbf{F} \Rightarrow \mathbf{G}$, is such that the following conditions hold:

- $\forall i \in N$, $I^i = \sigma(i)$, $I^i_{in} = \gamma(i)$, and $(I^i, I^i_{in} \stackrel{s}{\Rightarrow} J^i, J^i_{emt})$ is a timed local transition for transducer $\mathcal{T}^i_N$, $I^i_{db} = P(\mathbf{I}, i)$, where s is the clock value provided by the environment; if the network is *broadcasting*, then $\gamma^b(i) = J^e_{emt}$ while, if it is *communication free*, $\gamma^f(i) = J^i_{emt}$, with $J^i_{emt} \subseteq J^e_{emt}$;

- $(I^e, I^e_{in} \Rightarrow J^e, J^e_{emt})$ is a local transition for the environment, where $I^e_{in} = \bigcup_{i \in N} J^i_{emt}$, and t is the initial time value.

[1]Note that because of the considered languages, we are assured that local transitions always terminate in at most PTIME.

Informally, during a global transition all the nodes composing the network make simultaneously a local transition taking as input the associated tuples. A local transition for the environment is then executed, whose input is the set of tuples emitted by all the transducers (i.e., $I^e_{in} = \bigcup_{i \in N} J^i_{emt}$). In addition we assume that one global transition, in order to satisfy properties **S3′** - **S3″**, can start only after a certain amount Δ of physical time has elapsed after the end of the previous transition. As a final remark note that, since a global transition is composed by $|N|$ deterministic local transitions, and the communication is assumed to be reliable, also global transitions are deterministic. From this follows that any *rsfd* system $\mathcal{S}^{rsfd}$ is composed by *just one run*.

3.5 Example: Multi-core Processing

A synchronous transducer network can be instantiated on top of a multi-core machine in a straightforward way. Assume to have a machine with $n = 8$ cores. We can consider N to contains 8 identifiers, one for each core, i.e., $|N| = n$. Given as input a transducer $\mathcal{T}$ and a partition function P, we can hence instantiate 8 relational transducer each of them acting over a partition $P(\mathbf{I}, i)$ of $\mathbf{I}$. In this way we will have a transducer network composed by 8 transducers and emulating multi-core computation. Communication in this case can be really efficient since we can take advantage of the *shared memory model* of the architecture. In this setting, we hence have that the environment results to be an abstraction of the shared memory and acting as a concurrency controller, while each transducer is taking as input a partition of the shared memory. Intuitively, on a multi-core machine all the assumption that we impose on the system can be easily enforced: the external clock is a function of the system clock, while reliable communication can be easily build on top of shared memory.

3.6 Example: Distributed Processing

A more difficult exercise in respect to the previous multi-core example is to instantiate a transducer network on top of a physically distributed system. In fact, although all the transducer nodes can be directly mapped into physical nodes, it is not clear where to map the environment. This problem can be solved using two different – not necessarily disjoined – perspective: a *logical*, and a *physical* one. Logically, the environment can be seen as a communication middleware providing both networking and synchronization services. Physically, the environment can be mapped on a set of networking devices – e.g., switches – providing the network connectivity where a special software routine is used to synchronize the nodes. In physically distributed systems, communication is performed using *message passing* and therefore the environment is used to abstract all the low level details intrinsic in this model.

3.7 Discussion

From a practical perspective, not all the assumptions we made are easy to implement as in multi-core settings. Although some of them are realistic, other are less realistic and therefore can be relaxed. Among the more realistics, there is the reliability assumption of the communication medium, which, even in networked system, can be naturally enforced by using, for example, the TCP-IP transmission protocol. On the other side, it seems pretty unreasonable in networked systems to have an entity collecting all the emitted tuples. As already mentioned in the previous section, this is an high-level abstraction useful for reasoning about the distributed computation, although a realistic perspective for this abstraction exists.

Nevertheless, the assumption of simultaneity among local transitions can be in general relaxed: a transducer which has not been fired yet in the current round is randomly chosen and fired. A global transition is then accomplished when all the transducers have been fired exactly once. This is exactly how most parallel models work in practice [29, 16]: processes execute their computation and wait until all other processes have finished, i.e., a *synchronization barrier* has been reached. In such settings, in order to avoid situations in which no transducer is fired, we can impose a *fairness* condition on global transitions: each transducer in N is eventually fired at least once in each round.

3.8 Query Computability

Given a run ρ describing a configured synchronous network, we use the signature $out(s)$ to denote the set of facts output by all nodes at time s, i.e., $out(s) = \bigcup_{i \in N} I^i_{out}$ s.t. $I^i_{out} \in \rho^i(s)$. Indeed this definition models exactly how parallel data processing frameworks work, i.e., the output is maintained distributed on each node composing a cluster. We say that a synchronous transducer network is *quiescent* if a time value s' exists such that $\forall s'' > s', out(s') = out(s'')$, that is, a *quiescence state* is reachable so that the output is stable and not changing anymore. We name (ρ, s') the *quiescence point*. We assume the resulting output for a synchronous network to be the output at the quiescence point, denoted by $out(*)$. If the synchronous network is not quiescent, we say that the output instance is $\perp$. As a signature, we use $\mathcal{N}_{N,t,P}(\mathbf{I})$ to denote the output of the configured network $\mathcal{N}_{N,t,P}$ on input database instance $\mathbf{I}$. In practice, a transducer network initial state identifies also its output. Intuitively, this is because the system $\mathcal{S}^{rsfd}_{\mathcal{N}_{N,t,P},\mathbf{I}}$ is constituted by one and only one run, therefore, given an initial instance and a configuration, the output is uniquely determined.

Definition 3. Given an input and an output schema, respectively Υ_{db} and Υ_{out}, a partial mapping $\mathcal{Q}_{out} : inst(\Upsilon_{db}) \to inst(\Upsilon_{out})$ is *computable by a bulk synchronous transducer network* $\mathcal{N}$ if, for each initial database instance $\mathbf{I}$ over Υ_{db}, a configured specification $\mathcal{N}_{N,t,P}$ exists such that:

C1 if $\mathcal{Q}_{out}(\mathbf{I})$ is undefined, then $\mathcal{N}_{N,t,P}(\mathbf{I}) = \perp$

C2 if $\mathcal{Q}_{out}(\mathbf{I})$ is defined, then $\mathcal{N}_{N,t,P}(\mathbf{I}) = \mathcal{Q}_{out}(\mathbf{I})$

Because we assumed generic queries as building blocks of transducers, we have that the function $\mathcal{N}_{N,t,P}$ is generic for each time and space oblivious synchronous transducer network.

Some transducer network may depend on the number of nodes available in the distributed system or on a particular partition function. In order to formally define this, we first introduce what it means for a constrained system to be *convergent*:

Definition 4. Given a constrained specification ψ, and an input instance $\mathbf{I}$, we say that the class $\boldsymbol{\mathcal{S}}^{rsfd}_{\psi,\mathbf{I}}$ is *convergent* if for all ρ, $\rho' \in \boldsymbol{\mathcal{S}}^{rsfd}_{\psi,\mathbf{I}}$, $out(*) = out'(*)$.

Thus, a constrained system is convergent if, at the quiescent point, all its runs have the same output. We now say that a specification $\mathcal{N}$ is *network-independent* if, fixed a partition function P and an initial time value t, the class $\boldsymbol{\mathcal{S}}^{rsfd}_{\mathcal{N}_{t,P},\mathbf{I}}$ is convergent, or, in other words, $\mathcal{N}_{N,t,P}(\mathbf{I})$ computes the same query $\mathcal{Q}_{out}(\mathbf{I})$ for all networks N. Similarly, fixed a set of nodes N, and a partition function P, we say that a specification $\mathcal{N}$ is *time-independent* if $\boldsymbol{\mathcal{S}}^{rsfd}_{\mathcal{N}_{N,P},\mathbf{I}}$ is convergent, while, fixed t and P, $\mathcal{N}$ is *partition-independent* if instead $\boldsymbol{\mathcal{S}}^{rsfd}_{\mathcal{N}_{N,t},\mathbf{I}}$ is convergent. Finally, if a specification $\mathcal{N}$ is network-time-partition-independent, the class $\boldsymbol{\mathcal{S}}^{rsfd}_{\mathcal{N},\mathbf{I}}$ is convergent, i.e., all the possible runs in $\boldsymbol{\mathcal{S}}^{rsfd}_{\mathcal{N},\mathbf{I}}$ compute the same query $\mathcal{Q}_{out}(\mathbf{I})$. This is because, whichever configuration is selected, $\boldsymbol{\mathcal{S}}^{rsfd}_{\mathcal{N},\mathbf{I}}$ has a unique output. In such case, $\mathcal{Q}_{out}$ is said to be *distributively computable*, while $\mathcal{N}$ is said to be *independent*.

Example 2. Assume we want to compute a distributed version of the join of Example 1. We can implement such join using a broadcasting and inflationary bulk synchronous transducer network which emits one of the two relations, let's say T, and then joins R with the received facts over T. UCQ is again expressive enough, and the transducer network can be specified as follows:

Schema: $\Upsilon_{db} = \{R^{(2)}, T^{(2)}\}, \Upsilon_{emt} = \{S^{(2)}\}, \Upsilon_{out} = \{Q^{(3)}\}$
Program: $S_{emt}(u, v) \leftarrow T(u, v).$
$Q_{out}(u, v, w) \leftarrow R(u, v), S(v, w).$

The specification is clearly independent since the same output is obtained whichever configuration is selected.

Synchronous specifications have the required expressive power:

LEMMA 1. *Let $\mathcal{L}$ be a language containing UCQ and contained in Datalog$^\neg$. Every query expressible in $\mathcal{L}$ can be distributively computed in 2 rounds by a broadcasting, inflationary, and oblivious $\mathcal{L}$-transducer network.*

The above lemma permits us to draw the following conclusion: under the *rsfd* semantics, monotonic and non-monotonic queries behave in the same way: two rounds are needed in both cases. This is due to the fact that, contrary to what happens in the asynchronous case [6], starting from the second round we are guaranteed – by the reliability of the communication and the synchronous assumption – that every node will compute the query over the full initial instance.

3.9 Coordination and the CALM Conjecture

The *CALM conjecture* [21] specifies that a well defined class of distributed programs can be computed in an *eventually consistent*, *coordination-free* way: *monotonic programs*. CALM has been proven in the following form for asynchronous systems [6].

CONJECTURE 1. *A query can be distributively computed by a coordination-free transducer network if and only if it is expressible in Datalog.*

Surprisingly enough, the conjecture does not hold in *rsfd* settings under the broadcasting communication model. Before showing this, we have to introduce the definitions of *eventually consistent* systems and *coordination-free* specifications.

Definition 5. Given two specifications $\mathcal{N}, \mathcal{N}'$ such that $\Upsilon_{db} = \Upsilon'_{db}$ and $\Upsilon_{out} = \Upsilon'_{out}$, an initial instance $\mathbf{I}$, and two configurations (N, t, P), (N', t', P'). We say that two systems $\mathcal{S}^{rsfd}_{\mathcal{N}_{N,t,P},\mathbf{I}}$ and $\mathcal{S}^{rsfd}_{\mathcal{N}'_{N',t',P'},\mathbf{I}}$ are *eventually consistent* if $out(*) = out'(*)$.

Informally, two systems are eventually consistent if they have the same *database* and *output* schema, and the output instances at the (possibly different) quiescent points are equivalent. Note that we have employed the term "eventually" to highlight the fact that the two systems are required to agree just on the initial instance $\mathbf{I}$ and on the final output state, and not on the entire execution.

We can now introduce the definition of coordination-freeness as stated in [6] and adjusted to fit into our context.

Definition 6. An indipendent specification $\mathcal{N}$ is *coordination-free* if for every initial instance $\mathbf{I}$ a non-trivial configuration (N, t, P) exists s. t., if $\mathcal{N}^f$ is the communication-free version of $\mathcal{N}$ obtained by setting γ^f as the communication function, $\mathcal{S}^{rsfd}_{\mathcal{N}^f_{N,t,P},\mathbf{I}}$ and $\mathcal{S}^{rsfd}_{\mathcal{N}_{N,t,P},\mathbf{I}}$ are eventually consistent.

Under our setting, it turns out that even the specifications used to distributively compute non-monotonic queries can be coordination-free, as the next example shows.

Example 3. Let $\mathcal{Q}_{out}$ be the "emptiness" query of [7]: given a nullary database relation $R^{(0)}$ and a nullary output relation $T^{(0)}$, $\mathcal{Q}_{out}$ outputs T iff R is empty. The query is non-monotonic. A FO-transducer network $\mathcal{N}^b$ can be easily generated to distributively compute $\mathcal{Q}_{out}$: every node first emits R if its local partition is not empty, and then each node locally evaluates the emptiness of R. Since the whole initial instance is installed on every node when R is checked for emptiness, T is true only if R is actually empty on the initial instance. The complete specification follows.

Schema: $\Upsilon_{db} = \{R^{(0)}\}, \Upsilon_{mem} = \{\texttt{Ready}^{(0)}\}, \Upsilon_{emt} = \{S^{(0)}\},$
$\Upsilon_{out} = \{T^{(0)}\}.$
Program: $S_{emt}() \leftarrow R().$
$\texttt{Ready}_{ins}() \leftarrow \neg\texttt{Ready}().$
$T_{out}() \leftarrow \neg S(), \texttt{Ready}().$

Now, assume that (N, t, P) is a non-trivial configuration and let $\mathcal{N}^f$ be the communication-free version of $\mathcal{N}^b$. Clearly, whichever initial instance $\mathbf{I}$ we select, $\mathcal{S}^{rsfd}_{\mathcal{N}^b_{N,t,P},\mathbf{I}}$ and $\mathcal{S}^{rsfd}_{\mathcal{N}^f_{N,t,P},\mathbf{I}}$ are eventually consistent when, e.g., P installs the full instance on every node.

The result we have is indeed counterintuitive: how is it possible that the CALM principle holds in the more general asynchronous setting, while it does not hold in more restrictive settings such as *rsfd* systems? It turns out that both the communication model and the definition of coordination-freeness proposed in [6] are not broad enough to work in general for synchronous systems. In the next sections we will see that our definitions of coordination freeness (Def. 19) guarantee eventually consistent computation for those queries that don't rely on broadcasting in order to progress. That is, the discriminating condition for eventual consistency is not monotonicity per se, but the absence of a fact that must be sent to all the nodes of the network. Thus the CALM conjecture can indeed be proven also in our context, under a more general definition of coordination and a more flexible communication model.

4. HASHING TRANSDUCER NETWORKS

Following other parallel programming models such as MapReduce, in this section we are going to introduce *hashing transducers*: i.e., relational transducers equipped with a content-based communication model founded on hashing. More precisely, let Υ be a transducer schema. For each relation $R^{(a)} \in \Upsilon_{emt}$, we fix a subset of its terms as the *hash-key* (*key* in short) terms for that relation. W.l.o.g. we will then use the notation $R^{(k,a)}$ to refer to a relation R of arity a having the first k terms specifying its key [2]. As a notation, we associate to every transducer schema Υ a *key-set* $\mathcal{K} = \{R^{(k,a)} \mid R^{(a)} \in \Upsilon_{emt}$, and $k \leq a$ is the key of $R\}$ containing every relation R for which a key is defined. It is then appropriate to refine how a hashing transducer $\mathcal{T}$ is defined. $\mathcal{T}$ is hashing if defined by the tuple $(\mathcal{P}, \Upsilon, \mathcal{K})$, where $\mathcal{P}$ is the transducer program, Υ the schema, and $\mathcal{K}$ the transducer's key-set. With each transducer network we can now associate a *distributed hash mapping* $\mathcal{H} : inst(\Upsilon_{emt}) \rightarrow 2^N \backslash \emptyset$ such that given a family of *hash*

[2]Note that if a relation R has its keys scattered in the tuple (for example the first and third term) w.l.o.g. we can substitute all the occurrence of R with a new relation having all the keys in the first k terms.

functions H composed by unary hash functions $h : \mathbf{dom} \to N$ and a fact $R(\mathtt{u}_1, \ldots, \mathtt{u}_a)$ over a relation $R^{(a)} \in \Upsilon_{emt}$ we have:

$$\mathcal{H}(R(\mathtt{u}_1, \ldots, \mathtt{u}_a)) = \bigcup_{i \in 1..k} h(\mathtt{u}_i), \text{ with } R^{(k,a)} \in \mathcal{K} \quad (5)$$

Similarly to [31], we employ hash functions to deterministically obtain the location to which a tuple belongs from its key-term values specified in $\mathcal{K}$, while $\mathcal{H}$ maps each fact to the set of nodes to which it must be delivered. In the special case in which for a relation $R^{(a)}$ no key is set, we write $k = \infty$, and we have that for each fact $R(\mathtt{u}_1, \ldots, \mathtt{u}_a) \in I_R$, $\mathcal{H}(R(\mathtt{u}_1, \ldots, \mathtt{u}_a)) = \bigcup_{h \in H} h(\mathbf{dom})$, i.e., every tuple is addressed to all the possible nodes reachable by the hashing family H. We label an emission query as *broadcasting* if the head relation R is such that $k = \infty$. In the case in which all the relations in $\mathcal{K}$ have the key set to ∞, we say that $\mathcal{K}$ is *unbounded*, *bounded* if, instead, for no relation $k = \infty$. If, for every $R^{(k,a)} \in \mathcal{K}$, $k = a$, we refer to $\mathcal{K}$ as *maximal*. We can identify two possible kinds of hash families: *partitioned*, when $\bigcup_{h \in H} h(\mathbf{dom}) \subsetneq N$, and *non-partitioned* otherwise. In the former case we have that the set of nodes composing the network can be divided into two sets: the set of nodes to which data can be hashed, and therefore are actively involved in the computation; and the one which instead is composed by all the nodes in $N \setminus \bigcup_{h \in H} h(\mathbf{dom})$. These do not receive any emitted tuple and therefore provide zero support towards the calculation of the final result. In order to make each node aware that under partitioned hash families only a subset of N is actually actively receiving data, we assume that the $system$ relation `All` contains a value for every node in $\bigcup_{h \in H} h(\mathbf{dom})$.

We base the definition of the communication function γ on $\mathcal{H}$, in such a way that $\gamma(i)$ returns only the facts node i is responsible for. More concretely, we use γ^h to denote this new communication function: $\forall i \in \mathcal{H}(I^e_{emt}), \gamma^h(i) = \{R(\bar{\mathtt{u}}) \mid R \in \Upsilon_{emt} \text{ and } R(\bar{\mathtt{u}}) \in I^e_{emt}\}$. We then name this new type of synchronous transducer networks $(\mathcal{T}, \mathcal{T}^e, \sigma, \gamma^h)$ as *hashing*. Henceforth we will often use the name transducer network to actually denote a hashing transducer network (and hence we will drop the h suffix). The definition of communication-free transducer can now be slightly revisited: a hashing transducer network $\mathcal{N}$ is communication-free if, for all relations $R \in \Upsilon_{emt}$, if $R(\bar{\mathtt{u}})$ is a fact derived by an emission query of node i, $\mathcal{H}(R(\bar{\mathtt{u}})) = i$.

Remark: From a more practical perspective, our formalization can be used to express how data addressing is performed between the map and the reduce tasks in the MapReduce framework, although we allow the same record to be addressed simultaneously to multiple reducers[3]. The special case in which the key is empty can instead be modeled by assigning the broadcasting address to the tuple.

Example 4. The following program is the hashed version of the one of Example 2:

Schema: $\Upsilon_{db} = \{R^{(2)}, T^{(2)}\}, \Upsilon_{emt} = \{S^{(1,2)}, U^{(1,2)}\}$,
$\Upsilon_{out} = \{J^{(3)}\}$

Program: $S_{emt}(u, v) \leftarrow R(u, v).$
$U_{emt}(u, v) \leftarrow T(u, v).$
$J_{out}(u, v, w) \leftarrow S(u, v), U(u, w).$

In this new form, every tuple emitted over S and U is hashed on the first term, so that we are assured that at least a node exists in which each pair of joining tuples is issued. Note that such program can be translated in MapReduce in a natural way.

[3]This behavior can be avoided, and the one-to-one communication semantics restored, for example by employing surrogate keys.

By employing hashing as addressing model, we are able to specify when a rule is *evaluated on a proper instance*:

Definition 7. Let $\mathcal{T} = (\mathcal{P}, \Upsilon, \mathcal{K})$ be a hashing transducer, N a set of nodes, and H a family of hash functions. We say that a query q_R in $\mathcal{P}$ is *evaluated on a proper instance* if for every relation $P^{(a)} \in \Upsilon_{emt}$ appearing in the body of q_R, every fact $\mathbf{f}$ over P satisfies $\mathcal{H}(\mathbf{f}) \cap i \neq \emptyset$ is in I^i_P.

In other words, a query is evaluated on a proper instance if every fact hashed to a certain node, is actually also in its state. Nevertheless, we would like not only to known when we are evaluating proper rules, but also when a rule is *live*, i.e., it will eventually derive something, and when instead is *safe* to derive a new fact.

Definition 8. Let $\mathcal{T} = (\mathcal{P}, \Upsilon, \mathcal{K})$ be a hashing transducer. We say that a query q_R in $\mathcal{P}$ is *live* if for every relation $P^{(a)} \in \Upsilon_{emt}$ in the body of q_R, if with $P(\bar{\mathtt{u}})$ we denote a fact over P available in the current transducer state, we have that:

$$\bigcap_{P \in body(q_R)} \mathcal{H}(P(\bar{\mathtt{u}})) \neq \emptyset \text{ if } \bigcup_{P \in body(q_R)} P(\bar{\mathtt{u}}) \text{ satisfies } q_R \quad (6)$$

Informally, a rule is live if each instance over $emit$ relations able to satisfy a rule-body is available in at least one node. We can now see how by proving that a query is live and evaluated on a proper instance, we are able to state that a specification have at least the same possibilities to distributively derive a fact as the original query computed locally on a single node.

Definition 9. Let $\mathcal{T} = (\mathcal{P}, \Upsilon, \mathcal{K})$ be an hashing transducer. We say that evaluating a query q_R in $\mathcal{P}$ is *safe* if for every fact $\mathbf{f}$ over a relation $P^{(a)} \in \Upsilon_{emt}$ appearing negated in the body of q_R we have that:

$$\forall i \in N, i \in \mathcal{H}(\mathbf{f}) \quad (7)$$

A query is considered safe if every node evaluates every negated literal on a proper instance containing all the facts available in the network over that literal. If the evaluation of a query is not safe the correctness of a specification can be jeopardize, as the next example shows.

Example 5. Consider the emptiness query of Example 3 computed by an hashing transducer network, over a nonempty instance $\mathbf{I}$. Note that $\mathcal{K}$ is unbounded. Every non trivial configuration will output T true even if the initial instance is nonempty if we select a partitioned family H. In fact, if this is the case, at least a node exists which will have relation S empty when `Ready` is set to true.

The last two definitions basically project upon our model the *safety* and *liveness properties* of specifications over distributed systems [23] . Informally, the safety property is used to state that "*nothing bad will ever happen*", while the liveness property certifies that "*something good will eventually happen*". In our model the safety property can be restated as "*no wrong fact will ever be derived*", and this is exactly Definition 9, while the liveness property can be reformulated as "*some fact will eventually be derived*", and this is what Definition 8 specifies. In this new guise, as Example 5 showed, the safety property must be clearly enforced when non-monotonic queries exist in a program, which consequently must be evaluated only if it is safe to do so. Conversely, the liveness property must always be satisfied if we want to ensured that every result will actually be computed by a specification.

4.1 Shuffling Transducers

Given a hashing transducer network, we can directly apply hashing functions to $database$ relations, and, once all the initial tuples are hashed, the actual queries can then be applied. We refer to such type of transducers as *shuffling* [4]. Intuitively, when $\mathcal{K}$ is unbounded and H is non-partitioned, the same behavior described in the proof of Lemma 1 under the broadcasting communication model is obtained. Note however that unsafe specifications (same discussion can be done for the liveness property) can still be written if $\mathcal{K}$ is not unbounded.

4.2 Parallelization Strategy

Modeling communication by means of hashing, we are able to create different computational strategies by simply customizing the adopted hash functions and relation keys. It is then reasonable to define the pair $(\mathcal{K}, H)$ as *parallelization strategy*: by selecting a set of keys and a family of hash functions, the programmer basically selects how computation should be allocated over a transducer network. While we have already seen that $\mathcal{K}$ is embedded directly into the definition of a transducer, H can be added to the configuration parameters of transducers and transducer networks. Henceforth, for hashing transducers and transducer networks we will respectively consider configurations determined by tuples in the form (N, i, H) and (N, t, P, H), where, as usual, N is the set of nodes, i a node identifier and P a partition function. Thus, once fixed a set of keys $\mathcal{K}$, a network $\mathcal{N}$ is called *strategy-independent* if whatever H is chosen, $\mathcal{N}$ computes the same query. Every strategy-independent specification is tolerant to the fact that H is partitioned or not. On the other hand, a specification can be strategy-dependent because of the fact that H is partitioned. Finally, a hashing specification is called *independent* if it is altogether network, time, partition and strategy-independent. Definition 4 of convergence class of systems generalizes also over classes defined by hashing specifications.

By incorporating into the transducer definition the parameter $\mathcal{K}$, we have that multiple specifications can be produced by selecting different sets of keys. Let us denote with $\boldsymbol{\mathcal{N}}$ the class of transducer networks which can be generated by changing $\mathcal{K}$ in the parallelization strategy. How are the elements composing these classes related to each other? Intuitively, a wrong selection of the keys can result in wrong specifications as the next example shows.

Example 6. Consider the transducer network of Example 4. Assume that we chose the second term as key for both S and U. We can then incur in the situation in which a tuple is not derived because the joining facts are issued to two different nodes.

Despite this, we can define a subclass of $\boldsymbol{\mathcal{N}}$ wherein each specification has a *correct key-set* $\mathcal{K}$.

Definition 10. Let $(\mathcal{T}, \mathcal{T}^e, \sigma, \gamma^b)$ be an independent broadcasting transducer network where $\mathcal{T} = (\mathcal{P}, \Upsilon)$. Let $\mathcal{N} = (\mathcal{T}', \mathcal{T}^e, \sigma, \gamma^h)$ be a hashing specification where $\mathcal{T}'$ is derived from $\mathcal{T}$ by adding the unbounded key-set $\mathcal{K}$. Consider the class of hashing specifications $\boldsymbol{\mathcal{N}}$ generated from $\mathcal{N}$.

A set of keys $\mathcal{K}'$ is said to be *correct* if, for every initial instance $\mathbf{I}$ and configuration (N, t, P, H), $\mathcal{S}^{rsfd}_{\mathcal{N}_{N,t,P,H},\mathbf{I}}$ and $\mathcal{S}^{rsfd}_{\mathcal{N}'_{N,t,P,H},\mathbf{I}}$ are eventually consistent, where with $\mathcal{N}' \in \boldsymbol{\mathcal{N}}$ we indicate the specification defined from the transducer $\mathcal{T}'' = (\mathcal{P}, \Upsilon, \mathcal{K}')$.

We refer to the sub-class of $\boldsymbol{\mathcal{N}}$ composed by specifications with correct key-sets as *correct*, and write $\boldsymbol{\mathcal{CN}}$. Note that specifications in $\boldsymbol{\mathcal{CN}}$ are network-time-partition-independent, but not necessarily strategy-independent.

[4]This name is inspired to the MapReduce's shuffling phase [16] although their functionality is not quite the same.

4.3 Parallel Computation of Datalog$^\neg$ Queries

In Section 3 we have seen that if the transducer network is broadcasting, a large class of queries can be distributively computed. We start by defining what it means for a query expressed in a language $\mathcal{L}$ to be *distributively computable* by a hashing transducer network. We say that a query $\mathcal{Q}_{out}$ is *distributively computable* if a specification exists such that all the possible runs in $\boldsymbol{\mathcal{S}}^{rsfd}_{\mathcal{N},\mathbf{I}}$ having H non-partitioned compute the same query $\mathcal{Q}_{out}(\mathbf{I})$, whichever proper initial instances $\mathbf{I}$ is given. The reader should not be surprised by the following result:

PROPOSITION 1. *Let $\mathcal{L}$ be a query language. Every query expressible in $\mathcal{L}$ which can be distributively computed by a broadcasting $\mathcal{L}$-transducer network can also be distributively computed by a hashing $\mathcal{L}$-transducer network.*

PROOF. By definition every hashing transducer emulates a broadcasting transducer when $\mathcal{K}$ is unbounded and H is non-partitioned. □

It is now a straightforward exercise to show that the expressiveness result of the previous section applies also for hashing specifications. W.l.o.g. we will hereafter name every hashing transducer network where $\mathcal{K}$ is unbounded and H is non-partitioned as *broadcasting*.

Over all we are interested in independent specifications. We will then say that a query is *parallelly computable* if it is distributively computable by an independent specification. But which class of queries can be parallelly computed? Indeed we are interested in monotonic Datalog queries because of the CALM conjecture.

LEMMA 2. *Let $\mathcal{L} \subseteq$ Datalog. Every monotonic query expressible in $\mathcal{L}$ can be parallelly computed by an inflationary and oblivious $\mathcal{L}$-transducer network.*

PROOF. Let $\mathcal{Q}_{out}$ be a monotonic query, and $\mathbf{I}$ an instance over $\mathcal{D}_{in}$. A specification $\mathcal{N}_{N,t,P,H}$ can be created by shuffling the input instance and then applying the query, as described in the first case of Lemma 1. $\mathcal{K}$ is unbounded, H is non-partitioned, and $\mathcal{N}_{N,t,P,H}$ distributively computes $\mathcal{Q}_{out}$. We have to show that $\mathcal{N}$ is actually independent and hence $\mathcal{Q}_{out}$ is parallelly computed by $\mathcal{N}$. Assume N to be non trivial. Consider now the same specification but with a new configuration (N, t, P, H') where H' is partitioned. The liveness property continues to hold also with the new configuration: $\mathcal{K}$ is unbounded and every query is evaluated over the full initial instance starting from the second round by at least a node, whichever H' is chosen. Every fact $\mathbf{f}$ in $\mathcal{Q}_{out}(\mathbf{I})$ is then also in $\mathcal{N}_{N,t,P,H'}(\mathbf{I})$ since the query is monotonic. It remains to show that no wrong fact will be derived by $\mathcal{N}_{N,t,P,H'}, \mathbf{I}$. We have that the safety property doesn't hold for $\mathcal{N}_{N,t,P,H'}, \mathbf{I}$ – although note that it does hold for $\mathcal{N}_{N,t,P,H}, \mathbf{I}$ – however we have that every passive node will always have all the $emit$ predicates empty. Hence, when $\mathcal{Q}_{out}$ is applied, no fact will be derived since the query is monotone. As a consequence we have that passive nodes do not contribute in any way toward the computation of the final query, although they participate in the shuffling. □

However, it turns out that monotonic queries are not the only parallelly computable queries. In fact, also a class of non-monotonic queries is parallelly computable: *chained queries*.

Definition 11. Let $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ be a conjunction of literals defining the body of a query q_R. We say that two different relations $R_i(\bar{u}_i)$, $R_j(\bar{u}_j) \in body(q_R)$ are *chained* in q_R if either:

- $\bar{u}_i \cap \bar{u}_j \neq \emptyset$; or
- a third relation $R_k \in q_R$, different from R_i, R_j exists such that R_i is chained with R_j, and R_k is chained with R_j.

Definition 12. We say that a query $\mathcal{Q}_{out}$ is *chained* if for every sub-query $q_R \in \mathcal{Q}_{out}$, no relation $R_i \in q_R$ exists which is either nullary or not chained with any other relation $R_j \in q_R$.

Note that every positive query composed just by one non-nullary body atom is chained by definition. Chained queries are the only types of non-monotonic queries which can be evaluated independently of the choice of the H parameter.

LEMMA 3. *Let $\mathcal{L}$ be a language such that $\mathcal{L} \subseteq$ Datalog$^\neg$. Every chained query expressible in $\mathcal{L}$ can be parallelly computed by an inflationary and oblivious $\mathcal{L}$-transducer network.*

PROOF. We proceed as in the proof of Lemma 2. Let $\mathcal{Q}_{out}$ be the input query, and $\mathbf{I}$ an instance over $\mathcal{D}_{in}$. A transducer network $\mathcal{N}_{N,t,P,H}$ can be created by shuffling the input instance and then applying the query from the second round over the entire instance, as described in the second case of Lemma 1. $\mathcal{K}$ is unbounded, H is non-partitioned, and $\mathcal{N}_{N,t,P,H}$ distributively computes $\mathcal{Q}_{out}$. Assume again N to be non trivial. We are going to show that $\mathcal{N}$ is actually independent, i.e., it parallelly computes $\mathcal{Q}_{out}$. Consider the same specification but with a new configuration (N, t, P, H') where H' is partitioned. Since all rules are chained, the liveness property continue to hold also with the new configuration: $\mathcal{K}$ is unbounded and every query is evaluated over the full initial instance starting from the second round by at least a note, whichever H' is chosen. It remains to show that no wrong fact will be derived by $\mathcal{N}_{N,t,P,H'}, \mathbf{I}$. Also in this case, we have that the safety property doesn't hold for $\mathcal{N}_{N,t,P,H'}, \mathbf{I}$. Note, however, that every passive node at the second round will have all the $emit$ predicates empty. Hence, when the query is applied, no fact will be derived because a new fact, to be derived, must be at least unary (the query is chained) and a constant must exist in the instance of a positive literal (only syntactically safe queries are considered). But this is impossible since every $emit$ relation is initially empty. We can then conclude that the transducer network $\mathcal{N}$ parallelly computes $\mathcal{Q}_{out}$. □

Among the parallelly computable queries, we are interested in all the queries for which a bounded specification, parallelly computing them, exists. We name this kind of queries as *hashing*. Intuitively, hashing queries are the ones which can be parallelly computed without having to broadcast any fact to all the nodes reachable by the hashing functions in H. The reader should not be deluded into believing that every monotonic query is trivially a hashing one.

Example 7. Assume two relations $R^{(2)}$ and $T^{(1)}$, and the following query $\mathcal{Q}_{out}$ returning the entire instance of R if T is nonempty.

$$Q(u, v) \leftarrow R(u, v), T(\_).$$

The query is clearly monotonic. Let $\mathcal{T}$ be the following (shuffling) broadcasting UCQ-transducer program parallelly computing $\mathcal{Q}_{out}$.

Schema: $\Upsilon_{db} = \{R^{(2)}, T^{(1)}\}, \Upsilon_{emt} = \{S^{(\infty,2)}, U^{(\infty,1)}\}$,
$\Upsilon_{out} = \{Q^{(2)}\}$
Program:
$S_{emt}(u, v) \leftarrow R(u, v).$
$U_{emt}(u) \leftarrow T(u).$
$Q_{out}(u, v) \leftarrow S(u, v), U(\_).$

Assume now a bounded set of keys $\mathcal{K}$. We have that, whichever $\mathcal{K}$ we chose, the related specification might no more be convergent. To see why this is the case, consider an initial instance $\mathbf{I}$ and $\mathcal{K}$ maximal. Assume $\mathbf{I}$ such that $adom(I_R) \supset adom(I_T)$, and a configuration in which N is large. In this situation, it may happen that a nonempty set of facts over R is hashed to a certain node i, while no fact over T is hashed to i. Hence no tuple emitted to i will ever appear in the output, although they do appear in $\mathcal{Q}_{out}(\mathbf{I})$. Thus this transducer is not convergent being strategy-dependent.

However, a class of monotonic queries exists which is hashing.

LEMMA 4. *Let $\mathcal{L} \subseteq$ Datalog be a language. Every chained monotonic query expressible in $\mathcal{L}$ can be made hashing.*

Not surprisingly, also non-monotonic queries are not hashing in general.

Example 8. Consider the following inflationary FO-transducer computing the facts in T not in the transitive closure of R.

Schema: $\Upsilon_{db} = \{R^{(2)}, T^{(2)}\}, \Upsilon_{emt} = \{S^{(1,2)}, U^{(1,2)}, CS^{(1,2)}\}$,
$\Upsilon_{mem} = \{\texttt{Ready}^{(0)}\}, \Upsilon_{out} = \{Q^{(2)}\}$
Program:
$S_{emt}(u, v) \leftarrow R(u, v).$
$U_{emt}(u, v) \leftarrow T(u, v).$
$CS_{emt}(u, v) \leftarrow S(u, v).$
$CS_{emt}(v, w) \leftarrow S(u, v), CS(u, w).$
$\texttt{Ready}_{ins}() \leftarrow \neg\texttt{Ready}().$
$Q_{out}(u, v) \leftarrow U(u, v), \neg CS(u, v), \texttt{Ready}().$

In this form the transducer network is not correct for all hashing functions. It might be the case that facts are distributed unevenly among the nodes. In this situation, a node may end up deriving a new fact over CS after that all the other nodes have already finished their computation. This may result in the possibility that a fact over Q must be retracted. Intuitively, the problem is that negation is applied too early. In order to avoid this situation, nodes must *synchronize*: every node should notify every other node that it has locally terminated the computation of CS. When every node has terminated its local computation of the closure of R, and every node knows this, then Q can be safely evaluated. Clearly this pattern requires a non-bounded specification. On the other hand the same query can be correctly computed if every node has the entire instance locally installed. In this case we are in fact guaranteed that every node will apply negation over the complete transitive closure, but this is obtainable only by using an unbounded specification.

Indeed a class of non-monotonic queries which does not require broadcasting rules exists: *recursion-bounded chained* queries.

Definition 13. Given a Datalog$^\neg$ query $\mathcal{Q}_{out}$, we say that $\mathcal{Q}_{out}$ is *recursion-bounded* if, whichever stratification $\Pi_1, \ldots, \Pi_n$ we chose, $\Pi_1, \ldots, \Pi_{n-1}$ are non-recursive programs, while Π_n is expressed in $\mathcal{L}$, with $\mathcal{L} \subseteq$ Datalog$^\neg$.

Informally, recursion-bounded queries are non-monotonic queries in which recursion is only allowed in the latest stratum.

PROPOSITION 2. *Let $\mathcal{L} \subseteq$ Datalog$^\neg$. Every recursion bounded chained query expressible in $\mathcal{L}$ is hashing.*

Figure 1 depicts the taxonomy of the queries discussed so far.

Remarks: (1) The problem existing with non-recursion bounded queries clearly does not present itself for monotonic queries. In fact, even if tuples are hashed unevenly, no retraction can occur since the query is monotonic. (2) If a mechanism existed for which a recursive Datalog query could be rewritten in a non-recursive form, all chained non-monotonic queries would be non-broadcasting. The problem of determining if a given recursive query is equivalent to some non-recursive program is called the *boundedness* problem, and, unfortunately, it is undecidable [20].

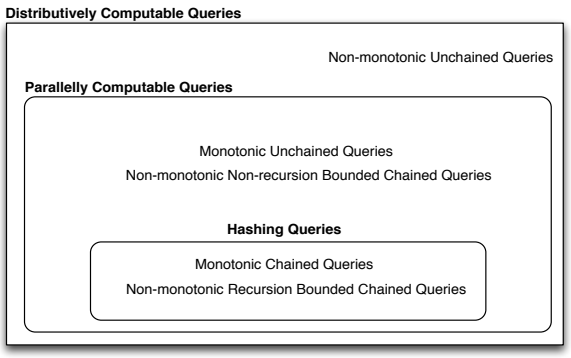

Figure 1: Queries' Taxonomy

5. REFINING COORDINATION-FREENESS

We have seen in Section 3.9 that, for synchronous and reliable systems (*rsfd*), a specific notion of coordination-freeness is needed. In fact we have shown that, under such model, certain non-monotonic queries – e.g., Example 3 – that require coordination under the asynchronous model can be computed in a coordination-free way. While in [6] coordination-freeness is directly related to communication-freeness, in this section we will see that this is not necessarily the case for synchronous and reliable systems.

5.1 Syncausality

Achieving coordination in asynchronous systems – i.e., systems where each process proceeds at an arbitrary rate and no bound exists on message delivery time – is a very difficult and costly task. A necessary condition for coordination in such systems is that some control over the ordering of events must be enforced by some primitives [22]. In a seminal paper [24], Lamport proposed a synchronization algorithm exploiting a relation of *potential causality* $\rightarrow$ over asynchronous events. According to Lamport, given two events e, e', we have that $e \rightarrow e'$ if e happens before e'[5] and e might have caused e'. From a high-level perspective, the potential causality relation models how information flows among processes, and therefore can be employed as a tool to reason on the patterns which cause coordination in asynchronous systems. A question now arises: what is the counterpart of the potential causality relation for synchronous systems? *Synchronous potential causality* (*syncausality* in short) as been recently proposed [9] to generalize Lamport's potential causality relation to synchronous systems. Using syncausality we are able to model how information flows among nodes with the passing of time. Given a specification class $\mathcal{N}$, and an input instance $\mathbf{I}$, if $\rho \in \mathcal{S}_{\mathcal{N},\mathbf{I}}$ is a run in the class of systems defined by $\mathcal{N}$, and (ρ^i, s), (ρ^j, s') are two points (cf. Section 3.2) for possibly not distinct nodes $i, j \in N$, we say that (ρ^j, s) causally depends on (ρ^i, s') if either $i = j$ and $s \leq s'$ – i.e., one point follows the other one locally on the same node – or a tuple has been emitted by node i at time s addressed to node j with $s < s'$[6]. We refer to these two types of dependencies as *direct*.

Definition 14. Given a system $\mathcal{S}$, and a run $\rho \in \mathcal{S}$, we say that two points (ρ^i, s), (ρ^j, s') are related by a *direct potential causality* relation, and write $(\rho^i, s) \rightarrow (\rho^j, s')$, if either:

1. $s' = s + 1$ and $i = j$;
2. $s' \geq s + 1$ and node i emitted a tuple at time s addressed to node j; or
3. $(\rho^i, s) \rightarrow (\rho^j, s')$ and $(\rho^j, s') \rightarrow (\rho^k, s'')$, then $(\rho^i, s) \rightarrow (\rho^k, s'')$.

Note that direct dependencies define precisely Lamport's happen-before relation – and hence we maintain the same signature $\rightarrow$.

Differently from asynchronous systems, we however have that a point (ρ^j, s') can occasionally *indirectly* depend on another point (ρ^i, s) if $\theta(s + 1) + var = \theta(s')$ and, even if no fact addressed to j is actually emitted by i, j can still draw some conclusion simply because of the *bounded delay guarantee* (**S3**) of synchronous systems. We can model the bounded delay guarantee as a virtual $NULL$ fact, similarly to [25]. Under this perspective, indirect dependencies appear the same as the direct ones, although, instead of a flow generated by "informative" facts, with the indirect relationship we model the flow of "non-informative", $NULL$ facts. The interesting thing about the bounded delay guarantee is that it can be employed to specify when the *Progressive Closed World Assumption* (PWCA) [33] holds and therefore negation can be safely applied to a predicate. In general, the PCWA holds for a negated literal $R(\bar{u})$ when the content of R is sealed for what concerns the current round and therefore negation can be applied. In local settings, we have that the PCWA holds for a predicate at round s' if its content has been generated at round s, with $s' > s$. In distributed settings, we have that if R is an $emit$ relation, being in a new round s' is not enough, in general, for establishing that its content is sealed. This is because tuples can still be floating, and therefore, until we are not assured that every tuple has been delivered, the PCWA does not hold. The result is that negation cannot be applied safely. We can reason in the same way also for every other negative literal depending on R. We will then model the fact that the content of an $emit$ relation R is stable because of the bounded delay guarantee **S3** – and hence the PCWA holds for it, or for dependent relations – by having every node i emit a fact $NULL_R^i$ at round s, for every $emit$ relation R, which will be delivered at node j exactly at time $\theta(s + 1) + var$. We then have that the content of R is stable once j has received a $NULL_R^i$ fact from every $i \in N$. The sealing of an $emit$ relation at a certain round is then ascertained only when $|N|$ $NULL_R$ facts have been counted. We call this *Snapshot Closed World Assumption* (*SCWA*): negation can be applied on a node just when it has surely received a correct snapshot of the global instance I_R. Note that not necessarily the $NULL_R^i$ facts must be physically sent. This in particular is true under the fixed delivery semantics, where the strike of a new round automatically seals all the $emit$ relations because of condition **S3**$''$. The reader however should not believe that this always holds. In fact, in Section 6.1, we will see how by simply dropping property **S3**$''$ the situation becomes more complicated. As a final remark, note that no $NULL_R^i$ fact will be issued if no SCWA need be enforced over R or over a relation depending on R.

Definition 15. Given a run $\rho \in \mathcal{S}^{rsync}$, we say that two points (ρ^i, s), (ρ^j, s') are related by an *indirect potential causality* relation, and we write $(\rho^i, s) \dashrightarrow (\rho^j, s')$, if $i \neq j$, $s' \geq s + 1$ and a $NULL_R^i$ fact addressed to node j has been emitted by node i at round s.

Example 9. Consider the hashing version of the program depicted in Example 3. Let (N, t, P, H) be a configuration and $\mathbf{I}$ a proper initial instance. At round $s + 1$ we have that the SCWA does hold for relation S, and hence negation can be applied. Note that if R is empty in the initial instance, no fact is emitted. Despite this, every node can still safely conclude at round $s + 1$ that the content of S is stable. In this situation we clearly have an indirect

[5]The potential causality relation is often referred to as *happen-before*.

[6]Note that a point in a synchronous system is what Lamport defined as event in an asynchronous system.

potential causality relation between the points (ρ^i, s), $(\rho^j, s+1)$, for all $i, j \in N$, with $\rho \in \mathcal{S}^{rsfd}_{\mathcal{N}_{N,t,P,H},\mathbf{I}}$.

We are now able to introduce the definition of syncausality: a generalization of Lamport's happen-before relation which considers not only the direct information flow, but also the flow generated by indirect dependencies.

Definition 16. Let ρ be a run in the system $\mathcal{S}$. The *syncausality* relation $\rightsquigarrow$ is the smallest relation such that:

1. if $(\rho^i, s) \rightarrow (\rho^j, s')$, then $(\rho^i, s) \rightsquigarrow (\rho^j, s')$;
2. if $(\rho^i, s) \dashrightarrow (\rho^j, s')$, then $(\rho^i, s) \rightsquigarrow (\rho^j, s')$; and
3. if $(\rho^i, s) \rightsquigarrow (\rho^j, s')$ and $(\rho^j, s') \rightsquigarrow (\rho^k, s'')$, then $(\rho^i, s) \rightsquigarrow (\rho^k, s'')$.

5.2 From Syncausality to Coordination

At the level of granularity we set so far, no clear pattern emerges from the syncausality structure of runs that can be linked to a proper notion of coordination. To solve this, we propose the *predicate-level syncausality* relationship.

Definition 17. Given a run $\rho \in \mathcal{S}$, we say that two points (ρ^i, s), (ρ^j, s') are linked by a relation of *predicate-level syncausality* over $R \in \Upsilon$, and write $(\rho^i, s) \overset{R}{\rightsquigarrow} (\rho^j, s')$, if any of the following holds:

1. $i = j$, $s' = s + 1$ and a tuple over $R \in \Upsilon_{mem} \cup \Upsilon_{out}$ has been derived by a query in $Q_{ins} \cup Q_{out}$;
2. $R \in \Upsilon_{emt}$ and node i emits a tuple over R at time s addressed to node j, with $s' \geq s + 1$;
3. $R \in \Upsilon_{emt}$ and node i emits a $NULL^i_R$ fact at time s addressed to node j, with $s' \geq s + 1$;
4. $(\rho^i, s) \overset{R}{\rightsquigarrow} (\rho^j, s')$ and $(\rho^j, s') \overset{R'}{\rightsquigarrow} (\rho^k, s'')$, then $(\rho^i, s) \overset{R,R'}{\rightsquigarrow} (\rho^k, s'')$; R, R' defines a *predicate chain*.

Intuitively, the predicate-level syncausality specifies a causal relation at the predicate level. The definition of *predicate-level direct* and *indirect potential causality* can be extracted from the predicate-level syncausality definition intuitivelly. We are now able to formally specify a condition for achieving coordination.

Definition 18. Let $\boldsymbol{\mathcal{CN}}$ be a correct class. We say that $\boldsymbol{\mathcal{CN}}$ manifests the *coordination pattern* if, for all possible initial instances $\mathbf{I} \in inst(\Upsilon_{db})$, whichever run $\rho \in \boldsymbol{\mathcal{S}}_{\boldsymbol{\mathcal{CN}},\mathbf{I}}$ we select in which N is not trivial, a point (ρ^i, s) and an $emit$ relation R exist so that $\forall j \in \bigcup_{h \in H} h(\mathbf{dom})$, there is a predicate-level syncausality relation such that $(\rho^i, s) \overset{R,\ldots}{\rightsquigarrow} (\rho^j, s+1)$ and $s + 1 \leq *$.

We call node i the *coordination master*. A similar pattern has been named *broom* in [10]. Informally, we have that coordination exists when a node sends a set of facts over a relation R to all the active nodes of the network. The condition $s + 1 \leq *$ is needed since only points which actually contribute to the definition of the final state are of interest, while all the others can be ignored. In practice, we have that a class is generated from a specification. The latter requires a coordination protocol if either the class is not correct – a coordination protocol is needed in order to make the specification independent – or the class is correct but no strategy exists which is not able to rise the coordination pattern – a coordination mechanism is already injected into the program.

If at least a non-trivial configuration exists so that the coordination pattern doesn't manifest itself, we have coordination-freeness:

Definition 19. Given a correct class $\boldsymbol{\mathcal{CN}}$ and an initial instance $\mathbf{I}$, let us denote with $\boldsymbol{\mathcal{S}}_{\boldsymbol{\mathcal{CN}},\mathbf{I}}$ the class of systems specified by $\boldsymbol{\mathcal{CN}}$. We then say that $\boldsymbol{\mathcal{CN}}$ is *coordination-free* if a parallelization strategy $(\mathcal{K}, H)$ exists such that a non-trivial configuration (N, t, P, H) can be selected for which, if ρ is a run in $\mathcal{S}_{\mathcal{N}_{N,t,P,H},\mathbf{I}}$, where $\mathcal{N}$ is the specification in $\boldsymbol{\mathcal{CN}}$ having $\mathcal{K}$ as set of keys, and i a node in N, $\mathcal{N}$ does not manifest the coordination pattern – i.e., no coordination master (ρ^i, s) exists.

W.l.o.g., we will dub specifications such as $\mathcal{N}$ coordination-free. From Definition 19 we can deduce the following proposition:

PROPOSITION 3. *Every coordination-free specification parallelly computes a hashing query.*

PROOF. If a specification is coordination-free, the only flow of information is the direct flow. In addition the direct flow must be such that a master node does not exist, i.e., no $emit$ relation is allowed to have a key set to ∞. □

The reverse result clearly does not hold: a hashing query might require coordination (non-monotonic chained recursion bounded queries are hashing and not coordination-free). In addition, since in every coordination-free specification broadcasting queries are not strictly necessary, we can deduce that at least a configuration exists – the trivial case is the configuration in which the partition function installs the full initial instance on one node, and H addresses every constant to the same node – for which the correct result is computed without emitting any fact, i.e., every coordination-free specification can be made communication-free.

PROPOSITION 4. *Let $\mathcal{N}$ be a coordination-free specification. For every initial instance $\boldsymbol{I} \in \Upsilon_{db}$ a not trivial configuration (N, t, P, H) exists so that $\mathcal{N}_{N,t,P,H}(\boldsymbol{I})$ is communication-free.*

PROOF. Let H so that a set of proper functions exists which hash every constant to the same node i. Similarly set P so that the full initial instance is installed on i, and N non-trivial. $\mathcal{N}_{N,t,P,H}(\mathbf{I})$ is convergent by definition and coordination-free. □

Example 10. As example of a communication-free specification consider the following UCQ-network $\mathcal{N}$ computing the transitive closure of the relation $R^{(2)}$: locally each node computes the closure of R on its local data and then emits the newly derived atoms.

Schema: $\Upsilon_{db} = \{R^{(2)}\}, \Upsilon_{emt} = \{S^{(1,1)}\}, \Upsilon_{out} = \{T^{(2)}\}$
Program:
$T_{out}(u, v) \leftarrow R(u, v).$
$T_{out}(v, w) \leftarrow S(u, v), T(u, w).$
$S_{emt}(u, v) \leftarrow T(u, v).$

$\mathcal{N}$ is oblivious and $\boldsymbol{\mathcal{S}}^{rsfd}_{\mathcal{N},\mathbf{I}}$ is convergent whichever initial instance $\mathbf{I}$ over Υ_{db} is given. Since there is no negation, we have only to show that for every initial instance, a configuration exists such that the emitting queries are not broadcasting. Consider P and H such that the full instance is installed on one node, and every constant is hashed to the same node. $\mathcal{N}_{N,t,P,H}$ is clearly communication-free.

We can therefore deduce that coordination-freeness is a sufficient but not necessary condition for a specifications to be communication-free, as shown by the next example.

Example 11. Let $\mathcal{Q}_{out}$ be the following non-monotonic query:

$$Q_{out}(v) \leftarrow R(u, v), \neg T(u).$$

The following is a FO-transducer network parallelly computing $\mathcal{Q}_{out}$:

Schema: $\Upsilon_{db} = \{R^{(2)}, T^{(1)}\}, \Upsilon_{mem} = \{\texttt{Ready}^{(0)}\}$,
$\Upsilon_{emt} = \{S^{(1,2)}, U^{(1,1)}\}, \Upsilon_{out} = \{Q^{(1)}\}$.
Program: $S_{emt}(u, v) \leftarrow R(u, v)$.
$U_{emt}(u) \leftarrow T(u)$.
$\texttt{Ready}_{ins}() \leftarrow \neg\texttt{Ready}()$.
$Q_{out}(v) \leftarrow S(u, v), \neg U(u), \texttt{Ready}()$.

The specification is non-monotonic, hashing, and can be made communication-free. Consider in fact a partition function P which installs the entire instance on a node i. Assume then H such that a hash function exists by which every emitted tuple is addressed to i, an arbitrary t and non-trivial N. Whichever initial instance $\mathbf{I}$ is given, we clearly have that $\mathcal{N}_{N,t,P,H}$ is communication-free.

We can hence state the following proposition which generalize Proposition 4 by properly defining the essence of hashing queries:

PROPOSITION 5. *Every hashing query is parallelly computable by a bounded specification which can be made communication-free.*

PROOF. By definition every hashing query is parallelly computable by a specification $\mathcal{N}$. We have to show that $\mathcal{N}$ can be made communication-free. If the query is monotonic we fall under the case of Proposition 4. We then consider here the case in which the query is non-monotonic. Fix a configuration where N is non-trivial, t is arbitrary, and P, H are such that the full initial instance is installed on node i, and a hash function exists in H so that every constant is hashed to the same node i. We have that $\mathcal{N}_{N,t,P,H}(\mathbf{I})$ is independent by definition, and communication-free. □

With Proposition 5 we described one of the characteristics of hashing queries: they are bounded and communication-free. In the next section, we will see that a larger class of queries can be computed in a communication-free way: *embarrassingly parallel queries* [7].

6. CALM IN RSYNC SYSTEMS

The original version of the CALM principle as postulated in Conjecture 1 is trivially not satisfiable in *rsfd* systems because a monotonic class of queries exists – i.e., unchained queries – which is not coordination-free as specified in Definition 19. We will then prove the CALM conjecture just for the remaining class of monotonic queries. We first introduce the following lemma:

LEMMA 5. *Let $\mathcal{L}$ be a query language. Every query that is parallelly computed by a coordination-free $\mathcal{L}$-transducer network is monotone.*

PROOF. Let $\mathcal{N}$ be a coordination-free $\mathcal{L}$-transducer network parallelly computing a query $\mathcal{Q}_{out}$. Let $\mathbf{I}$ and $\mathbf{J}$ be two initial instances over $\mathcal{D}_{in}$ such that $\mathbf{I} \subseteq \mathbf{J}$. We have to show that $\mathcal{Q}_{out}(\mathbf{I}) \subseteq \mathcal{Q}_{out}(\mathbf{J})$. Assume first that $adom(\mathbf{J} \setminus \mathbf{I}) \cap adom(\mathbf{I}) = \emptyset$. Since $\mathcal{N}$ parallelly compute $\mathcal{Q}_{out}$, $\mathcal{N}$ is convergent, therefore a non-trivial configuration exists such that P installs $\mathbf{I}$ on one single node, while $\mathbf{J} \setminus \mathbf{I}$ is completely installed in another node. In addition, assume an hash function in H such that all the constant in $\mathbf{I}$ are hashed to the same node in which $\mathbf{I}$ is installed, while all the constant in $\mathbf{J} \setminus \mathbf{I}$ are hashed to the node in which $\mathbf{J} \setminus \mathbf{I}$ resides. Consider a fact $\mathbf{f} \in \mathcal{Q}_{out}(\mathbf{I})$. Indeed $\mathbf{f}$ will appear in the output of $\mathcal{N}_{N,t,P,H}(\mathbf{I})$ at node i at a certain round s. Consider now the case in which the input instance is $\mathbf{J}$. We will have a node $j \neq i$ such that $I^j_{db} = \mathbf{J} \setminus \mathbf{I}$, while again $I^i_{db} = \mathbf{I}$. Let us consider the point (ρ, s) of the run $\rho \in \mathcal{S}^{rsfd}_{\mathcal{N}_{N,t,P,H},\mathbf{I}}$. Because local transitions are deterministic, and no fact in $\mathbf{J} \setminus \mathbf{I}$ can be addressed from j to i, $\mathbf{f}$ is output again also in run ρ. Again, being $\mathcal{N}$ convergent, $\mathcal{N}_{N,t,P,H}(\mathbf{J})$ parallelly computes the query $\mathcal{Q}_{out}(\mathbf{J})$, therefore $\mathbf{f}$ belongs also to $\mathcal{Q}_{out}(\mathbf{J})$.

Consider now the specification $\mathcal{N}^f$ builded from $\mathcal{N}$ by using γ^f instead of γ^h. We can seamlessly use this procedure since $\mathcal{N}$ is communication-free by Proposition 4. We then have that $\mathcal{N}_{N,t,P,H}(\mathbf{I}) = \mathcal{N}^f_{N,t,P,H}(\mathbf{I})$ and, similarly, $\mathcal{N}_{N,t,P,H}(\mathbf{J}) = \mathcal{N}^f_{N,t,P,H}(\mathbf{J})$. Consider now two generic input instances $\mathbf{J}', \mathbf{I}'$ with $\mathbf{J}' \supseteq \mathbf{I}'$, and the same partition function P installing $\mathbf{I}'$ on node i and $\mathbf{J}' \setminus \mathbf{I}'$ on node j. Also in this case we have that $\mathcal{N}^f_{N,t,P,H}$ parallelly computes $\mathcal{Q}_{out}$ and, for the same reasoning of above, $\mathcal{N}^f_{N,t,P,H}(\mathbf{I}') \subseteq \mathcal{N}^f_{N,t,P,H}(\mathbf{J}')$ so $\mathcal{N}^f$ is monotonic. As a consequence also $\mathcal{N}$ is monotonic. □

We are now able to prove the (restricted version of the) CALM conjecture for *rsfd* systems:

THEOREM 1. *A query can be parallelly computed by a coordination-free transducer network if and only if chained and expressible in Datalog.*

PROOF. Starting from the if direction, by Lemma 2 we know that a chained Datalog query can be parallelly computed by an oblivious hashing transducer network $\mathcal{N} \in \boldsymbol{\mathcal{CN}}$. It remains to show that $\mathcal{N}$ is coordination-free. We can notice that, being the transducer network monotonic, no coordination pattern can occur because of indirect information flow (a necessary condition for an indirect flow to exists is the presence of a negated literal). On the other side it may occurs because of direct information flow caused by broadcasting rules. Let $\mathcal{N}' \in \boldsymbol{\mathcal{CN}}$ a specification (not necessarily different from $\mathcal{N}$) such that $\mathcal{K}$ is bounded. Note that a bounded specification exists because every chained Datalog query is hashing by Lemma 4. It remains to show that a non trivial configuration (N, t, P, H) exists, such that for every initial instance $\mathbf{I}$, a run $\rho \in \mathcal{S}^{rsfd}_{\mathcal{N}_{N,t,P,H},\mathbf{I}}$ exists in which the coordination pattern doesn't appear. Let us assume H to contain a hash function such that every fact emitted is always addressed to the same node. Indeed the coordination pattern cannot exists in ρ since all the tuples are addressed to the same node. Finally, $\mathcal{N} \in \boldsymbol{\mathcal{CN}}$, therefore $\mathcal{N}_{N,t,P,H}$ correctly computes the query. For what concern the only-if direction – although it is well known that a set of monotonic queries exists which are not expressible in Datalog [3] – it is covered by Lemma 5. □

From the previous section we know that every coordination-free specification $\mathcal{N}$ is indeed convergent, i.e., $\mathcal{N} \in \boldsymbol{\mathcal{CN}}$. But what about communication-freeness? As a preliminary answer to this question we can try to give a different reading of the CALM principle, by relating communication-freeness (instead of coordination-freeness) with monotonicity.

PROPOSITION 6. *Every oblivious specification parallely computing a Datalog query can be made communication-free.*

PROOF. Assume that a proper initial instance $\mathbf{I}$ is given. Consider first a coordination-free specification $\mathcal{N}$ computing a hashing monotonic query. We have already seen in Preposition 4 that a configuration exists which makes $\mathcal{N}$ communication-free.

Consider now the case in which the monotonic specification $\mathcal{N}$ is computing a query $\mathcal{Q}_{out}$ which is not hashing. Consider a configuration (N, t, P, H) as described in Preposition 4: P installs the

[7] The term embarrassingly parallel comes from the parallel computing field, where it refers to the class of problems parallelly solvable by a set of tasks, without resorting to communication [19].

full instance on a unique node i, H addresses all the emitted facts to i, and N, t are arbitrary (but N is not trivial). Let $\mathcal{N}^f_{N,t,P,H}, \mathbf{I}$ be the fully instantiated communication-free version of $\mathcal{N}$. We have to show that $\mathcal{N}^f_{N,t,P,H}(\mathbf{I})$ computes the query $\mathcal{Q}_{out}(\mathbf{I})$. Consider first all the nodes $j \neq i$ in N. Such nodes will not output anything since their instance is empty and the query is monotone. For what concern i, it exactly computes $\mathcal{Q}_{out}(\mathbf{I})$ since every query is evaluated on a proper instance. □

By leveraging Propositions 5 and 6 we have that a wide class of queries can be parallelly computed in a communication-free way. As previously mentioned, we name this class *embarrassingly parallel*.

Definition 20. Let $\mathcal{L}$ be a language and $\mathcal{Q}_{out}$ a $\mathcal{L}$-query. $\mathcal{Q}_{out}$ is *embarrassingly parallel* if and only if it is parallelly computable by a specification which can be made communication-free.

We can now state the following *EPIC* (*Embarrassingly Parallel Is Communication-free*) Theorem:

THEOREM 2. *Let $\mathcal{L}$ be a query language containing UCQ. For every query $\mathcal{Q}_{out}$ expressible in $\mathcal{L}$, the following are equivalent:*

1. *$\mathcal{Q}_{out}$ can be parallelly computed by a communication-free transducer network;*
2. *$\mathcal{Q}_{out}$ can be parallelly computed by an oblivious, inflationary transducer network; and*
3. *$\mathcal{Q}_{out}$ is embarrassingly parallel.*

PROOF. $3 \Rightarrow 2$ from Lemma 2 (monotonic case) and Lemma 3 (non-monotonic case); $1 \Rightarrow 3$ by Definition 20. It remain to prove that every oblivious and inflationary transducer can be made communication-free. We already know from Proposition 5 and 6 that the only kind of queries which can be parallelly computed and are neither coordination-free nor communication-free are the non-recursion bounded queries. We proceed by contradiction: assume $\mathcal{N}$ is a non-monotonic transducer network computing the query $\mathcal{Q}_{out}$, and $\mathcal{Q}_{out}$ is not recursion-bounded, i.e., a recursive stratum m exists, which is followed my another stratum $m+1$. By definition of stratification the stratum $m+1$ cannot be evaluated before stratum m has terminated or wrong facts could be derived. The transducer, in order to correctly compute the query, must therefore be able to detected when the recursion is terminated and hence the evaluation of the $m+1$ stratum can start. Since each node composing the network could ends up having different partitions of the initial instance, every node may terminate the recursive computation in different rounds. Note that although a partition might exist for which recursion terminates at the same round for every node, $\mathcal{N}$ is convergent by definition, therefore it must be able to compute $\mathcal{Q}_{out}$ even in the case in which recursion terminates unevenly. Every node can detect that every other node has terminated its local recursive computation only by a direct information flow. In particular, a broadcasting communication must be executed since every node must communicate to every other active node that it has finished its local computation. To express that a node have finished its computation, `id` must be clearly read, otherwise every receiving node would not be able to identify which node has actually terminated the computation. Every node, in addition, in order to deduce that every other node has terminated its local computation, must read the `All` relation to count that every node in the network has communicated that the local computation is completed. Clearly this is not an oblivious specification since $system$ relations are employed. This conclude the proof, since every parallelly computable oblivious query is either monotonic or non-monotonic recursion bounded and chained, that are exactly the queries which can be made communication-free. □

As a result of the above theorems, we have that three different classes of coordination patterns can be identified under the *rsfd* semantics: *snapshot coordination*, which is obtainable by exploiting indirect information flow and hence is communication-free for *rsfd* systems, and is used by any non-monotonic recursion bounded chained query; *broadcasting coordination*, implementable by simple broadcasting queries and required for monotonic unchained queries; and, finally, *synchronized coordination*, which necessarily requires access to $system$ relations, since non-monotonic non-recursion-bounded queries must be synchronized by a direct information flow in order to maintain convergence.

An interesting difference about the three coordination patterns is that the first two depend on the system's semantics, while synchronized coordination is tolerant to any changes over the distributed system model. In the next section we will hence show how the semantics of snapshot and broadcasting coordination patterns changes if we weaken the constraint of the system definition. So far, in fact, we have only considered the fixed delivery model, i.e., tuples arrive exactly at time $\theta(s+1)+var$ if emitted at round s. But what would happen if we assume less constrained systems? In the next sections we will first take a look at systems with *non-deterministic delivery* but bounded variance (condition **S3**$''$ of Section 3.3 doesn't hold); and then we will conclude with *rsync* systems, i.e., systems with non-deterministic delivery and arbitrary finite variance. Figure 2 updates the raffiguration of parallelly computable queries of Figure 1 with the new results just discussed.

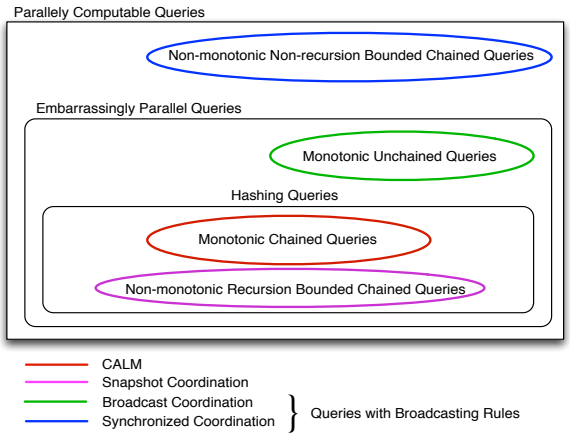

Figure 2: Complete Taxonomy for Parallelly Computable Queries under *rsfd*

6.1 Coordination and Non-deterministic Delivery

Given a synchronous transducer network $\mathcal{N}$, if we assume a synchronous system with non-deterministic delivery $\mathcal{S}^{rsbv}$, we have that different behaviors arise based on the kind of transducer program. In fact, let us consider first the case in which the program is monotonic: in this circumstance no wrong tuple can be derived by definition, even if an emitted fact is received after the round has already started. Therefore monotonic transducer networks behave equivalently under *rsbv* and *rsfd* systems. The same thing cannot be stated for non-monotonic programs, as the next example shows.

Example 12. Let us assume we have the usual transducer parallelly computing the emptiness query over a nullary relation R, as

described in Example 3. Since we are now under non-deterministic delivery, it is not clear when negation can be safely applied over S. If negation is immediately applied, it may happen that a new fact appears later, therefore invalidating the derived results.

In synchronous systems with non-deterministic delivery we have that snapshot coordination is no more achievable in general "indirectly", without exchanging any message. Under this model we can therefore appreciate more in detail the nature of snapshot coordination. In order to explain how snapshot coordination can be implemented in *rsbv* systems, we first solve the problem under the constraint that the communication means[8] implement *First In First Out* (*FIFO*) delivery. We will then consider the general case.

6.1.1 Snapshot Coordination under FIFO

Under the FIFO assumption we have that tuples are received in the same order in which they are locally derived[9]. Recall that snapshot coordination – implementing the SCWA – is used to ascertain that a relation is sealed. We can reduce the problem of detecting the sealed state of a relation to the problem of detecting a global stable property in a distributed system [8], and therefore apply one of the well-known snapshot protocols working under FIFO [13].

Let $\mathcal{T}$ be a transducer used to parallelly compute a non-monotonic recursion-bounded query $\mathcal{Q}_{out}$. In Section 5.2 we have seen that $NULL$ messages are implicitly sent by emission queries under the fixed delivery assumption. With non-deterministic delivery we could be forced to explicitly send $NULL$ messages. Consider a relation R occurring negated in the body of a query in $\mathcal{Q}_{out}$. Let $\mathbf{B}$ be the body of the query $q_R \in \mathcal{Q}_{emt}$ emitting R. We can add to Q_{emt} the following queries defining respectively a unary *stratified aggregate* relation $\texttt{CntR}^{(1)}$ [10], and a new unary $emission$ relation $\texttt{NullR}^{(\infty,1)}$ emulating the $NULL$ message for R:

$$\texttt{CntR}(count < \bar{u} >) \leftarrow \mathbf{B}. \tag{8}$$

$$\texttt{NullR}_{emt}(i) \leftarrow \texttt{CntR}(u), \texttt{Id}(i). \tag{9}$$

In this way, by exploiting the stratified semantics, we are able to emit the $NULL$ message for relation R soon after the computation of the count of the number of tuples in R is completed. Since count is a stratified aggregate, `CntR` – and therefore also `NullR` – belongs to a higher stratum. In this way we are assured that the $NULL$ message is emitted after that the instance over R is completed. Under the FIFO semantics we are then guaranteed that once a node receives a `NullR` tuple, the content of R is sealed for what concerns that emitting node. This clearly doesn't mean that R is globally sealed, since a tuple produced by a different node can still be floating. To have the SCWA hold on R, a node must have received a number of $NULL$ messages equal to the number of nodes composing the network: i.e., negation is applied on a stable snapshot of R. To obtain this, we add to $\mathcal{T}$ the rules:

$$\texttt{CntNullR}(fs\_count < u >) \leftarrow \texttt{NullR}(u). \tag{10}$$

$$\texttt{CntAll}(count < u >) \leftarrow \texttt{All}(u). \tag{11}$$

$$\texttt{SCR}() \leftarrow \texttt{CntNullR}(u), \texttt{CntAll}(u). \tag{12}$$

and we attach `SCR()` to the queries in which R is negated. Here we have employed two different types of aggregate functions: stratified aggregates, with the usual semantics, and *monotonic aggregates* as defined in [27], labeled with the fs (*frequency support*) prefix. Differently from stratified aggregates always returning a single value – i.e., the application of a function over a stable multi-set of values – for monotonic aggregation, a monotonically evolving distribution of values is instead returned, based on the evolution of the instances defining the body. Queries (10) - (12) are hence used to define when `SCR` is true, i.e., when n $NULL$ messages have been received for relation R, with n the number of active nodes in the network. Once `SCR` is true, negation can be safely applied over R, so the related query can be evaluated (if no other negative literal appears in the same query). Note how we have employed monotonic and stratified aggregates: since we do not know when `NullR` is stable, we cannot apply to it stratified aggregates nor negation, while we can definitely use a stratified aggregate over `All`. Interestingly, just by moving from a *rsfd* system to a *rsbv* system, both $system$ relations must be employed to implement snapshot coordination, therefore non-monotonic chained recursion bounded queries are no more embarrassingly parallel. This is consistent with [7]: non-monotonic queries are neither coordination nor communication-free, and both `Id` and `All` relations are required. In non-deterministic delivery systems, we then have that syncausality degenerates into the Lamport's happen-before relation [24]. Figure 3 depicts this new situation in which snapshot coordination code is injected in the specification.

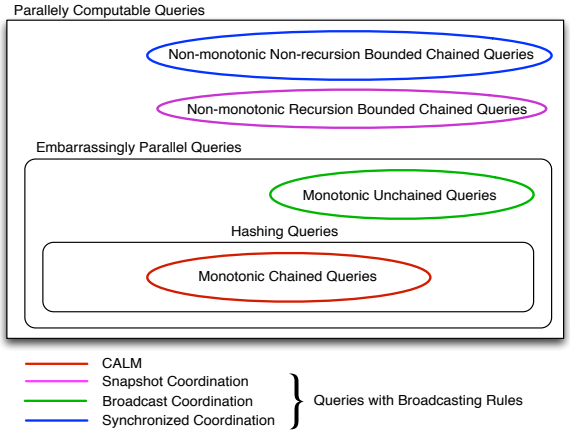

Figure 3: Taxonomy for Parallelly Computable Queries under *rsbv*

6.1.2 Generic Snapshot Coordination

If we drop the FIFO assumption, we could end up in a situation in which a $NULL$ message is received before a regular tuple, therefore negation can result in being applied on a non-stable relation. Indeed this problem is related to how a partial-order of events can be enforced in distributed settings [24]. However, in our specific case, we are not interested in a complete ordering of the emitted tuples over R, but, instead, we just want to be able to state that `SCR` is true when n $NULL$ messages have been received *and* no other tuple over R is still floating. More concretely, we are interested in implementing just the *gap-detection property* [8] of ordered events: we want to be able to determine if, once received an event (the $NULL$ message) there is some other event (emitted tuple) happened before it, which has not been received yet. Negation cannot, in fact, be applied until we are not guaranteed that our knowledge base has no gap. To implement this, query (9) can be modified as follows:

$$\texttt{NullR}_{emt}(i, u) \leftarrow \texttt{CntR}(u), \texttt{Id}(i). \tag{9.a}$$

[8] By communication means here we intend both the node's local buffer and the actual communication medium.

[9] Note that this doesn't mean that tuples that are derived on different nodes in a certain global order are also received in the same order. FIFO can therefore be seen as enforcing a partial order.

[10] Here and in the next sections we employ aggregate relations in the head of queries in the form $R(\bar{u}, \Lambda < w >)$, with Λ one of the usual aggregate functions, w a variable from the body, and $\bar{u}$ a list thereof, called the grouping variables.

where NullR is now a binary relation containing also the number of tuples originally emitted for R. Before applying (10) - (12) we have to ensure that the number of tuples over R is equal to the number of tuples originally emitted. We then add to $\mathcal{T}$ the clauses:

$$\texttt{CntR}(fs\_count < \bar{u} >) \leftarrow R(\bar{u}). \quad (13)$$

$$\texttt{SmNR}(w) \leftarrow \texttt{SmNR}(u), \texttt{NullR}(i, v), w = u + v. \quad (14)$$

$$\texttt{SmNR}(0). \quad (15)$$

counting the number of tuples in R and the total number of tuples over R derived globally, and finally we modify eq. (12) as follows:

$$\texttt{SCR}() \leftarrow \texttt{CntNullR}(u), \texttt{NAll}(u), \texttt{SmNR}(v), \texttt{CntR}(v). \quad (12.a)$$

In this way we are ensured that negation can be applied over R only if the proper number of $NULL$ messages is received and, at the same time, all the emitted R-facts have been received.

Remark: Note how our implementation of snapshot coordination makes precise the intuition that (non-monotonic) coordination requires waiting and that "waiting requires counting" [21].

6.2 Coordination under Arbitrary Variance

If we assume that also condition **S3**$'$ does not hold, we have that the variance can be arbitrary long although finite. Surprisingly, in this situation we have that monotonic unchained queries become coordination-free. To see why this is the case, first remark that a fact emitted at round s is still delivered at most at time $\theta(s+1) + var$, but, since var is now arbitrary, we are no more assured that $\theta(s+1) \leq \theta(s+1) + var < \theta(s+2)$. Despite this, the notion of coordination still maintains its semantics, even if, in this case, the coordination pattern may span multiple rounds. Let $\mathcal{N}$ be a specification parallelly computing a monotonic unchained query, $\mathbf{I}$ an instance, and (N, t, P, H) a non-trivial configuration. Under the *rsync* semantics we have that the system $\mathcal{S}^{rsync}_{\mathcal{N}_{N,t,P,H},\mathbf{I}}$ is composed by multiple eventually consistent runs, one for each possible round $\theta^{-1}(\theta(s+1) + var)$ in which a fact emitted at round s is actually received. A configuration (N, t, P, H) can then be chosen – e.g., the one where P installs the entire initial instance $\mathbf{I}$ on every node – so that no coordination pattern arise because the final state is already reached without having any broadcasted fact been received.

Example 13. Consider the monotonic unchained query of Example 7. Assume a *rsync* specification $\mathcal{N}$ defined as in Example 7 but where now $S^{(2,2)}$ and $\mathcal{Q}_{out}$ contains also the query: $Q_{out}(u, v) \leftarrow S(u, v), T(\_)$. Consider the non-trivial configuration (N, t, P, H) in which P installs the full instance $\mathbf{I}$ on every node, while $\mathcal{H}(I_S) \subset \bigcup_{h \in H} h(\mathbf{dom})$ – i.e., the instance over S is not broadcasted. We then have that a run $\rho \in \mathcal{S}^{rsync}_{\mathcal{N}_{N,t,P,H},\mathbf{I}}$ exists such that every fact emitted over S at round s is received by round $s' \leq *$, while every fact over U that should be sent to a node in $\bigcup_{h \in H} h(\mathbf{dom}) \setminus \mathcal{H}(I_S)$ is received in a round $s'' > *$. Clearly, we still have that the correct output is returned since I_T is composed by at least one fact, and every emitted tuple over S has been correctly received. The class defined by $\mathcal{N}$ is coordination-free.

We can therefore conclude that the original CALM principle is fully satisfied under the *rsync* semantics since every monotonic (Datalog) query can now be computed in a coordination-free way. The reader can now completely appreciate how the notion of coordination we introduced perfectly aligns with the one introduced in [6] when arbitrary variance comes into play: embarrassingly parallel queries are all coordination-free, and hence CALM and EPIC coincide. Our definition is then more general since it can be seamlessly used in both synchronous and asynchronous systems. Figure 4 shows the new taxonomy when *rsync* systems are considered.

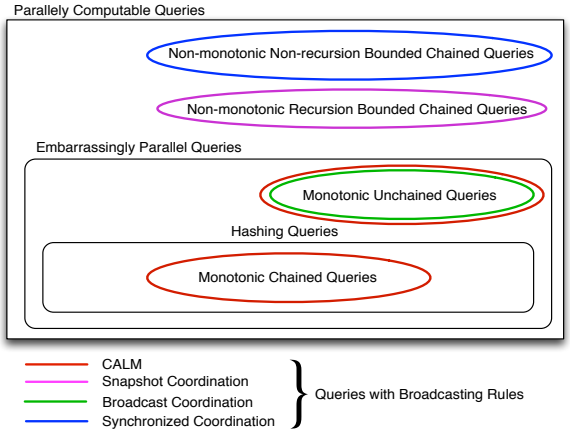

Figure 4: Taxonomy for Parallelly Computable Queries under *rsync*

Remark: From a states of knowledge perspective, in *rsfd* systems (*rsbv*, *rsync*), *common knowledge* (*δ-common knowledge*) can be obtained by simply using broadcasting [18]. However, if the final outcome is returned before δ-common knowledge is reached, the former was computed without coordination. For what concerns non-monotonic queries, the result of a query can be correctly computed only if the stability of the negated predicates is common knowledge among the nodes in the network. This highlights the main difference between broadcasting and snapshot/synchronized coordination: the former exists in *rsfd* - *rsbv* just because of the tight requirements imposed on the system; the latter are required by the actual semantic of the query.

7. RELATED WORKS

Our computational model merges the original transducer network model [7] – representing how distributed computation is carried out by an asynchronous system – with the Bulk Synchronous Parallel model [29]. We chose to embed BSP into the transducer network model to be able to formally represent computation in data-parallel systems, since many Big Data frameworks can considered directly [26] or indirectly [28] as an implementation of the BSP model. The concept of environment that we have employed has been adopted from the multi-agent system domain [18]. Thanks to the environment, we are not only able to model concerns not directly expressed into programs but that could affect their semantics – such as the global clock – but also provide a logical counterpart for the *master node* commonly found in data-parallel systems.

In this paper we advocate that CALM should be employed as a basic theoretical tool also for synchronous systems. The reader could be induced to believe that CALM indeed would not hold in synchronous settings, since such kind of systems already embed some notion of coordination. As shown in [9, 10] this is clearly not true if a formal definition of coordination is taken into consideration, i.e., coordination viewed as a particular state of knowledge required to obtain a shared agreement in a group of nodes. From this perspective our work is thus addressing a complementary domain with respect to *Bloom* [4, 15].

In Bloom, programs are statically analyzed and *points of order* identified: i.e., code positions where non-monotonic logic exists which could bring inconsistent outcomes [4]. From our perspective, point of orders identify where an indirect information flow exists. We favor *point of stability* as a better label for identifying

these circumstances, since *stability* of the input facts is what is actually required. Based on the cause of inconsistency, in [15] two different coordination strategies have been identified: *sealing* and *ordering*. Both of them are comparable to our snapshot coordination: the former in the general case; the later in a global version of the FIFO case, where they obtain global ordering by exploiting external coordinations systems such as Zookeper [22]. In addition, we have identified broadcasting and synchronized coordination.

Finally, note that the CALM principle in its original form is satisfied only if every node is not permitted access to any information on how data was originally partitioned. In this case, in fact, certain weaker forms of monotonic programs can be evaluated in a coordination-free way [34, 5]. From our point view, this is possible because the data-partitioning is already common knowledge before the computation starts, i.e., nodes already embed some notion of coordination. We deem this as a realistic assumption in certain specific scenarios. We therefore plan to extend our present work in order to consider also weaker forms of monotonicity.

8. CONCLUSIONS

In this paper the CALM principle is analyzed under synchronous and reliable settings. By exploiting CALM, in fact, we would be able to break the synchronous cage of modern parallel computation models, and provide asynchronous coordination-free executions when allowed by the program logic. In order to reach our goal, we have introduced a new abstract model emulating data-parallel computation, and a novel interpretation of coordination which generalize the one employed by Ameloot [6].

REFERENCES

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

[2] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 179–187, 1998.

[3] F. Afrati, S. S. Cosmadakis, and M. Yannakakis. On datalog vs. polynomial time. *J. Comput. Syst. Sci.*, 51(2):177–196, Oct. 1995.

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

[5] 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. In *To appear in Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '14, New York, NY, USA, 2014. ACM.

[6] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 283–292, New York, NY, USA, 2011. ACM.

[7] T. J. Ameloot, F. Neven, and J. Van Den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2):15:1–15:38, 2013.

[8] O. Babaoğlu and K. Marzullo. Distributed systems (2nd ed.). chapter Consistent global states of distributed systems: fundamental concepts and mechanisms, pages 55–96, 1993.

[9] I. Ben-Zvi and Y. Moses. Beyond lamport's *happened-before*: On the role of time bounds in synchronous systems. In N. A. Lynch and A. A. Shvartsman, editors, *DISC*, volume 6343 of *Lecture Notes in Computer Science*, pages 421–436. Springer, 2010.

[10] I. Ben-Zvi and Y. Moses. On interactive knowledge with bounded communication. *Journal of Applied Non-Classical Logics*, 21(3-4):323–354, 2011.

[11] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, 2011.

[12] E. A. Brewer. Towards robust distributed systems. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[13] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[14] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010.

[15] I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, editors. *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. IEEE, 2014.

[16] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[18] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, Cambridge, MA, USA, 2003.

[19] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[20] H. Gaifman, H. Mairson, Y. Sagiv, and M. Y. Vardi. Undecidable optimization problems for database logic programs. *J. ACM*, 40(3):683–713, July 1993.

[21] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[22] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010.

[23] E. Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272, 1994.

[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[25] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2):254–280, Apr. 1984.

[26] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[27] M. Mazuran, E. Serra, and C. Zaniolo. Extending the power of datalog recursion. *VLDB J.*, 22(4):471–493, 2013.

[28] M. F. Pace. Bsp vs mapreduce. *Procedia Computer Science*, 9(0):246 – 255, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

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

[30] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[31] O. Wolfson and A. Ozeri. A new paradigm for parallel and distributed rule-processing. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pages 133–142, New York, NY, USA, 1990. ACM.

[32] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[33] C. Zaniolo. Logical foundations of continuous query languages for data streams. In P. Barceló and R. Pichler, editors, *Datalog*, volume 7494 of *Lecture Notes in Computer Science*, pages 177–189, 2012.

[34] D. Zinn, T. J. Green, and B. Ludäscher. Win-move is coordination-free (sometimes). In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pages 99–113, 2012.

APPENDIX

A. ADDITIONAL PROOFS

Proof of Lemma 1: This is an application to our context of Theorems 4.9 - 4.10 in [7]. Let $\mathcal{Q}_{out}$ be a query expressed in $\mathcal{L}$ having input schema $\mathcal{D}_{in}$, and output schema $\mathcal{D}_{out}$. We have two cases, (i) $\mathcal{Q}_{out}$ is monotone or (ii) is not. In case (i), we can program a transducer $\mathcal{T}$ so that initially all nodes send out their local database facts. In the next round all nodes will have received all database facts because communication is synchronous and reliable. Relations `Id` and `All` are not needed. The query $\mathcal{Q}_{out}$ is then evaluated on the union of the local initial partition with the received facts.

More precisely, we have that the transducer schema will have $\Upsilon_{db} = \mathcal{D}_{in}$, $\Upsilon_{emt} = \{(R', a) \mid R^{(a)} \in \Upsilon_{db}\}$, $\Upsilon_{out} = \mathcal{D}_{out}$ while $system$ and $time$ schemas are as usual. Denote with $\mathcal{Q}'_{out}$ the version of the query where all the edb relations are primed. The transducer program $\mathcal{P}$ is then composed by the set of queries Q_{emt} and Q_{out}, where Q_{emt} is composed by a set of rules in the form:

$$R'_{emt}(\bar{u}) \leftarrow R(\bar{u}). \tag{16}$$

one for each $R \in \Upsilon_{db}$. On the other side Q_{out} simply contains $\mathcal{Q}'_{out}$. The network is clearly monotonic and oblivious.

In case (ii) we follow the same approach as previously, but this time the query $\mathcal{Q}_{out}$, being non-monotonic, cannot be applied immediately because wrong results could be derived. To avoid this, let us assume $\mathcal{T}$ is the transducer of case (i). We then add to Υ_{mem} the nullary relation `Ready`, and to Q_{ins} the query:

$$\texttt{Ready}_{ins}() \leftarrow \neg\texttt{Ready}(). \tag{17}$$

In addition we AND-attach to all the rules composing $\mathcal{Q}'_{out}$ the relation `Ready`. In this way, the query $\mathcal{Q}_{out}$ will be evaluated just starting from the second round since for the first one `Ready` is false. We therefore reach our goal.

Example 14. Let $\mathcal{Q}_{out}$ be the following UCQ query:

$$T(u, v) \leftarrow P(u, v), R(u).$$

A UCQ-transducer computing the same query is:

Schema: $\mathcal{T}_{db} = \{(R, 1), (P, 2)\}, \mathcal{T}_{emt} = \{(R', 1), (P', 2)\}$, $\mathcal{T}_{out} = \{(T, 2)\}$

Program:
$$R'_{emt}(u) \leftarrow R(u).$$
$$P'_{emt}(u, v) \leftarrow P(u, v).$$
$$T_{out}(u, v) \leftarrow P'(u, v), R'(u).$$

Example 15. Let $\mathcal{Q}_{out}$ be the following (non-monotonic) non-recursive Datalog$^\neg$ query:

$$T(u, z) \leftarrow R(u, v), P(v, z).$$
$$Q(u, z) \leftarrow S(u, v), \neg T(v, z).$$

with $\mathcal{D}_{in} = \{R^{(2)}, P^{(2)}, S^{(2)}\}$ and $\mathcal{D}_{out} = Q^{(2)}$. A non-recursive Datalog$^\neg$ transducer computing the same query is:

Schema: $\mathcal{T}_{db} = \{(R, 2), (P, 2), (S, 2)\}, \mathcal{T}_{mem} = \{(\texttt{Ready}, 0)\}$, $\mathcal{T}_{emt} = \{(R', 2), (P', 2), (S', 2)\}, \mathcal{T}_{out} = \{(Q, 2)\}$

Program:
$$R'_{emt}(u, v) \leftarrow R(u, v).$$
$$P'_{emt}(u, v) \leftarrow P(u, v).$$
$$S'_{emt}(u, v) \leftarrow S(u, v).$$
$$\texttt{Ready}_{ins}() \leftarrow \neg\texttt{Ready}().$$
$$T(u, z) \leftarrow R'(u, v), P'(v, z), \texttt{Ready}().$$
$$Q(u, z)_{out} \leftarrow S'(u, v), \neg T(v, z), \texttt{Ready}().$$

Proof of Lemma 4: From Lemma 2 we know that a specification $\mathcal{N}$ exists parallelly computing every monotonic query. Starting from $\mathcal{N}$ we can construct a new specification $\mathcal{N}'$ where every query in Q_{out} is primed and moved to Q_{emt}, and every auxiliary relation (if exists) is made $emit$. We then add to Q_{out} a rule to output every fact over $\mathcal{D}_{out}$. The behavior of $\mathcal{N}'$ is very simple: every time a new fact is derived by a rule, it is shuffled. We have that the liveness property is naturally enforced also in $\mathcal{N}'$ because $\mathcal{K}$ is unbounded and, for this reason, every query is evaluated on a proper instance and live. Every fact in $\mathcal{Q}_{out}(\mathbf{I})$, and no more, must hence also be in $\mathcal{N}'(\mathbf{I})$ whichever configuration we chose since the query is monotonic.

Consider $\boldsymbol{\mathcal{CN}'}$ the class of correct specifications defined by $\mathcal{N}'$. Assume now a new specification $\mathcal{N}''$ derived by $\mathcal{N}'$ by considering a bounded set of keys $\mathcal{K}''$. For simplicity we fix $\mathcal{K}''$ to be maximal. We have to show that $\mathcal{N}''$ is in $\boldsymbol{\mathcal{CN}'}$, i.e., it parallelly computes the same query of $\mathcal{N}'$. This is quite straightforward: every query is chained so the liveness property still hold because $\mathcal{K}''$ is maximal an hence a non-null intersection exists between the destination of all the atoms composing every rule-body. For the same reasoning, every query is also evaluated on a proper instance. For what concern the safety property, this is not an issue since the query is monotonic.

We are now going to make the proof more precise. An inflationary transducer $\mathcal{T} = (\mathcal{P}, \Upsilon, \mathcal{K})$ can be created where $\mathcal{K}$ is maximal and every query in $\mathcal{Q}_{out}$ is an emission query. Every derived fact over $\mathcal{D}_{out}$ is then output. Since every rule is chained we have the same opportunity in deriving facts as in the case in which $\mathcal{K}$ is unbounded. Because of the liveness property, and adding the fact that no wrong results can be derived since the query is monotone, the transducer network $(\mathcal{T}, \mathcal{T}^e, \sigma, \gamma^h)$ parallely computes $\mathcal{Q}_{out}$. $\mathcal{Q}_{out}$ is therefore hashing.

More precisely, let $\Upsilon_{db} = \mathcal{D}_{in}$, $\Upsilon_{emt} = \{R' \mid R \in sch(\mathcal{Q}_{out})$, $\Upsilon_{out} = \mathcal{D}_{out}$ ($systems$ and $time$ relations are as usual), and every term in every $emit$ relation is key. Since $\mathcal{Q}_{out}$ is monotonic, no result can be derived which will be retracted in the future. The idea is to apply every rule as it is, and every new derived fact is emitted to the other nodes composing the network. Concretely, we first add to Q_{emt} for each $R \in \Upsilon_{db}$ the following rule in order to implement the shuffling:

$$R'_{emt}(\bar{u}) \leftarrow R(\bar{u}). \tag{18}$$

Now, let $\mathcal{Q}'_{out}$ the version of $\mathcal{Q}_{out}$ where every relation is primed. We add to Q_{emt} all the rules in $\mathcal{Q}'_{out}$, and to Q_{out} a rule:

$$R_{out}(\bar{u}) \leftarrow R'(\bar{u}). \tag{19}$$

for each relation $R \in \mathcal{D}_{out}$ in order to output the results. Note that the transducer is oblivious and monotonic.

We now prove iteratively that a transducer program generated in this way indeed compute the initial query $\mathcal{Q}_{out}$. Let $\mathbf{I}$ be a proper instance over $\mathcal{D}_{in}$, and (N, t, P, H) a configuration. Assume $\mathbf{f}$ to be a fact over the relation $R \in \mathcal{D}_{out}$ and appearing in $\mathcal{Q}_{out}(\mathbf{I})$. We are going to show that $\mathbf{f} \in \mathcal{N}_{N,t,P,H}(\mathbf{I})$. We proceed by contradiction: we assume that $\mathbf{f} \notin \mathcal{N}(\mathbf{I})$ and we show that this is impossible. Consider the quiescent point (ρ, s) of $\mathcal{S}^{rsfd}_{\mathcal{N}_{N,t,P,H},\mathbf{I}}$. Being $\mathbf{f} \notin \mathcal{N}_{N,t,P,H}(\mathbf{I})$, by rule (19) we have that the related primed fact $\mathbf{f}'$ is not in the instance of the relation R' at round $s-1$. This means that at round $s-2$ an emission rule r' exists in Q_{emt} which has not derived the fact $\mathbf{f}'$, i.e., the body of the rule was not satisfied. Iteratively repeat this reasoning until an emission rule is reached whose body contains more that one relation. If such rule doesn't exist, $\mathbf{f}'$ was not derived because an extensional tuple $\mathbf{e}$ was not in the initial instance. Clearly this is a contradiction since $\mathbf{f}$ is in $\mathcal{Q}_{out}(\mathbf{I})$ and hence also $\mathbf{e}$ must be in $\mathbf{I}$. Assume then that an emission rule exists

having more that on relation in the body. W.l.o.g. we can assume that this rule is not satisfied because a fact $\mathbf{f}''$ over the $emit$ relation R'' was not in the local instance of R'' and, in the case it was, the rule would be satisfied. This may have occurred for three reasons: (i) $\mathbf{f}''$ was addressed to a wrong node; (ii) $\mathbf{f}''$ was never derived by the respective rules in the previous round, or (iii) $\mathbf{f}''$ was derived previously but was not persisted. Case (iii) is trivially impossible since the transducer is inflationary. Case (ii) is also impossible since every emitted tuple is hashed over all of its terms and, being the query chained and composed by more than one relation, at least a literal $R'''(\bar{v})$ exists which is chained with $R''(\bar{u})$, and contains an instance that will join with $\mathbf{f}''$. Since such instance must exists by construction, and it has been correctly hashed, because $\bar{u} \cap \bar{v} \neq \emptyset$, if instances over R''' are correctly addressed, also every instance over R'' must be addressed to the correct node. We can then conclude that $\mathbf{f}''$ was never derived. Repeating iteratively the same procedure one can show that $\mathbf{f}$ was not derived because a base fact $\mathbf{e}$ was not in the initial instance. We have already see that this is impossible, therefore $\mathbf{f}$ must be in $\mathcal{N}_{N,t,P,\mathcal{H}}(\mathbf{I})$. On the other hand it can be easily seen that every fact in $\mathcal{N}_{N,t,P,\mathcal{H}}(\mathbf{I})$ must also be in $\mathcal{Q}_{out}(\mathbf{I})$ since $\mathcal{T}$ is composed by the same rules forming $\mathcal{Q}_{out}$.

Example 16. Let $\mathcal{Q}_{out}$ be the usual transitive closure query.

$$T(u,v) \leftarrow R(u,v).$$
$$T(u,w) \leftarrow R(u,v), T(v,w).$$

A Datalog transducer parallely computing the same query is:

Schema: $\mathcal{T}_{db} = \{R^{(2)}\}, \mathcal{T}_{emt} = \{S^{(2,2)}, U^{(2,2)}\}, \mathcal{T}_{out} = \{T^{(2)}\}$

Program:
$$S_{emt}(u,v) \leftarrow R(u,v).$$
$$U_{emt}(u,v) \leftarrow S(u,v).$$
$$U_{emt}(u,w) \leftarrow S(u,v), U(v,w).$$
$$T_{out}(u,v) \leftarrow U_{emt}(u,v).$$

Proof of Proposition 2: We follows the same procedure presented in the proof of Lemma 4. Let $\Pi_1, \ldots, \Pi_n$ be a stratification of the rules of the input query $\mathcal{Q}_{out}$. An inflationary transducer $\mathcal{T} = (\mathcal{P}, \Upsilon, \mathcal{K})$ can be created where $\mathcal{K}$ is maximal and every stratum is evaluated sequentially. Every derived fact over $\mathcal{D}_{out}$ is then output. Since all the rules are chained, and the set of keys is maximal, the liveness property is always satisfied so we have the same opportunity in deriving fact as in the case in which $\mathcal{K}$ is unbounded. Although the safety property is not satisfied, no wrong result can be inferred because each stratum is evaluated sequentially and every rule is chained end evaluated on a proper instance. We can then conclude that the transducer network $(\mathcal{T}, \mathcal{T}^e, \sigma, \gamma^h)$ parallely compute $\mathcal{Q}_{out}$, and $\mathcal{Q}_{out}$ is an hashing query.

More precisely, let initially set $\Upsilon_{db} = \mathcal{D}_{in}$, $\Upsilon_{emt} = \{R' \mid R \in sch(\mathcal{Q}_{out})$, $\Upsilon_{out} = \mathcal{D}_{out}$ ($systems$ and $time$ relations are as usual), and every term in every schema relation is key. We start to generate $\mathcal{P}$ by adding to Q_{emt} a the rule in the form of eq. (18) for each $R \in \Upsilon_{db}$ to implement the shuffling (similarly to the proof of Lemma 4). Since $\mathcal{Q}_{out}$ is non-monotonic, a proper order of evaluation of rules must be enforced if we don't want to derive wrong results. Thus, consider the stratification $\Pi_1, \ldots, \Pi_n$ of $\mathcal{Q}_{out}$. First consider the first $n-1$ strata. By definition of recursion bounded query, such strata are non-recursive, therefore, looking at the predicate dependency graph, we can assign a set of predicates, inside the same stratum, to a *stage*. This assignment follows the dependency graph, so that each predicate who depends on another predicates belongs to an higher stage. Intuitively, the stratification is maintained since any predicates belonging to an higher stratum also belong to an higher stage. Let m the higher stage so obtained. We create a new stage $m+1$ containing all the predicates in the the last stratum Π_n. Consider now the query $\mathcal{Q}'_{out}$ obtained from $\mathcal{Q}_{out}$ (i) by priming all relations and (ii) by appending to the body of each rule in Π_j a nullary atom $\texttt{Stage}^j()$, with $j \in 1, \ldots, m+1$, in order to bind the evaluation of rules with the respective stage. We now add to Q_{emt} all the queries in $\mathcal{Q}'_{out}$, and to Q_{out} a rule in form of eq. (19) for each $output$ relation. Finally, in order to advance stage by stage we add one rule in the form:

$$\texttt{Stage}^j_{ins}() \leftarrow \texttt{Stage}^i(). \tag{20}$$

for each $0 < i < j < n$ and a rule:

$$\texttt{Stage}^1_{ins}() \leftarrow \neg\texttt{Stage}^1(), \neg\texttt{Stage}^2(), \ldots, \neg\texttt{Stage}^n. \tag{21}$$

to define when the first stage can start.

We are now going to prove that the transducer network derived from $\mathcal{P}$ actually computes the initial query $\mathcal{Q}_{out}$. The main difference in respect to the proof of Lemma 4 is that now the query is non-monotonic and therefore each negative literal cannot be evaluated before all the related tuples are generate by the lower strata. Let us proceed inductively: initially all the stages are false and only the rules implementing the shuffling can be evaluated. In the successive super-step, the first stage is active. Now, all the queries having just extensional relations of $\mathcal{Q}_{out}$ as body literal are evaluated. Let denote with q_R one of such rules. Since every previously emitted fact is hashed over all the terms composing the tuple, and since every rule is chained, we have that at least a node which is able to satisfy $body(q_R)$ exists, and a proper set of facts will be emitted. No wrong tuple can be derived because every rule is chained. All the new derived intensional facts, plus the previous extensional tuples are now emitted to a set of nodes based on the parallelization strategy. The query of the successive stage will then be evaluated in the successive round, and again, by construction, they are all evaluated on a proper instance. Let now assume we are at stage m and that ever query is evaluated sequentially on a proper instance until that point. Again this means that a new set of facts will be emitted, together with the previously emitted ones. At stage $m+1$ every rules is clearly still correctly evaluated. Note that the $m+1$-th stage can take more than one round to produce all tuples since it allowed to be recursive. We finally have that every rule in Q_{out} is evaluated on a proper instance by construction. This conclude the proof.

Example 17. Let $\mathcal{Q}_{out}$ be the following query:

$$T(u,v) \leftarrow E(u,v), \neg F(u).$$
$$T(u,w) \leftarrow E(u,v), T(v,w).$$

The above query basically is a transitive closure applied over a filtered set of edges. A Datalog transducer parallely computing the same query is the following:

Schema: $\mathcal{T}_{db} = \{E^{(2)}, (F^{(1)})\}, \mathcal{T}_{emt} = \{S^{(2,2)}, U^{(1,1)}, T^{(2,2)}\},$
$\mathcal{T}_{mem} = \{(\texttt{Stage}^1, 0), (\texttt{Stage}^2, 0)\}, \mathcal{T}_{out} = \{Q^{(2)}\}$

Program:
$$\texttt{Stage}^1_{ins}() \leftarrow \neg\texttt{Stage}^1(), \neg\texttt{Stage}^2().$$
$$\texttt{Stage}^2_{ins}() \leftarrow \texttt{Stage}^1().$$
$$S_{emt}(u,v) \leftarrow E(u,v).$$
$$U_{emt}(u) \leftarrow F(u).$$
$$T_{emt}(u,v) \leftarrow S(u,v), \neg U(u), \texttt{Stage}^1().$$
$$T_{emt}(v,w) \leftarrow S(u,v), T(u,w), \texttt{Stage}^2().$$
$$Q_{out}(u,v) \leftarrow T(u,v).$$

B. FURTHER EXAMPLES

In this section we give some additional examples of hashing transducer networks with aggregation.

Example 18. As a first example let us consider a simple task which applies a filter (left-join) to a relation and then computes the number of occurrences of the second term for each instance of the first one. The above task can be implemented locally by the following Datalog$^\neg$ program Π:

$$S(u,v) \leftarrow R(u,v), F(u).$$
$$T(u, count < v >) \leftarrow S(u,v).$$

where $edb(\Pi) = \{R^{(2)}, F^{(1)}\}$, $idb(\Pi) = \{S^{(2)}, T^{(2)}\}$ and R is the relation the filter F must be applied upon. Π can be computed in two rounds by the following Datalog$^\neg$ transducer (assuming a partition function P in which I_F is installed on every node):

Schema: $\Upsilon_{db} = \{R^{(2)}, F^{(1)}\}, \Upsilon_{emt} = \{R^{(1,2)}, F^{(1,1)}, S^{(1,2)}\},$
$\Upsilon_{emt} = \{S^{(1,2)}\}, \Upsilon_{mem} = \{\texttt{Ready}^{(0)}\}, \Upsilon_{out} = \{T^{(2)}\}$
Program: $\texttt{Ready}_{ins}() \leftarrow \neg\texttt{Ready}().$
$S_{emt}(u,v) \leftarrow R(u,v), F(u).$
$T_{out}(u, count < v >) \leftarrow S(u,v), \texttt{Ready}().$

As can be seen, the distributed and local implementations are practically identical [11]. In addition, the same task can be implemented in MapReduce in a natural way, for example by having the *map* function computing the filter and the hashing and the *reduce* function the aggregate.

Example 19. The previous example, although correct, in some case can be inefficient, e.g., when the filter predicate is composed by one single value and therefore all the values are addressed to one single node to compute the count. Indeed the same problem exists also in the MapReduce implementation. In such case, the *combiner* function can be exploited in order to obtain partial counts at the mapper side. The *reduce* function then can just sum up all the partial counts. Using our model we can easily achieve the same behavior. While the transducer schema remains as in Example 18, the program is now:

$$\texttt{Ready}_{ins}() \leftarrow \neg\texttt{Ready}().$$
$$S_{emt}(u, count < v >) \leftarrow R(u,v), F(u).$$
$$T_{out}(u, sum < v >) \leftarrow S(u,v), \texttt{Ready}().$$

Example 20. If we still focus our attention on the transducer network of Example 18, we can see that just changing a little bit the transducer definition we can obtain a network dependent transducer. In fact, if instead of the first term, we set the key of the $emit$ relation S to be the second term, clearly the correct sum cannot be obtained for all the possible sets of nodes since the tuples are grouped over the wrong term (v instead of u).

Example 21. Let us assume now we have just one relation $R^{(1)}$ and the following program, which returns the number of facts locally stored into R:

Schema: $\Upsilon_{db} = \{R^{(1)}\}, \Upsilon_{mem} = \{\texttt{Ready}^{(0)}\},$
$\Upsilon_{emt} = \{(S^{(1,1)}\}, \Upsilon_{out} = \{T^{(1)}\}$
Program: $\texttt{Ready}_{ins}() \leftarrow \neg\texttt{Ready}().$
$S_{emt}(u) \leftarrow R(u).$
$T_{out}(count < u >) \leftarrow S(u), \texttt{Ready}().$

The specification is intuitively not convergent: if we have a non trivial configuration and a nonempty initial instance, a family H exists containing at least an hash function h such that the number of facts in at least one node in N is greater or smaller than the number of facts stored in the other nodes.

Example 22. In this example we show how the counting of the number of paths between nodes in a directed acyclic graph can be computed in a parallel way [12].

Schema: $\Upsilon_{db} = \{R^{(2)}\}, \Upsilon_{emt} = \{S^{(1,2)}\}, \Upsilon_{out} = \{T^{(3)}\},$
Program: $P(u,v, count < (u,v) >) \leftarrow R(u,v).$
$T_{out}(v,u,c) \leftarrow P(u,v,c).$
$T_{out}(v,z, sum < (c) >) \leftarrow S(u,v,c1),$
$T(u,z,c2), c = c1 * c2.$
$S_{emt}(v,u,c) \leftarrow T(u,v,c).$

[11] Note though that F must be installed on every node. This can be accomplished by selecting a proper partitioning function P, or by adding a rule broadcasting F before any S-fact is emitted.

[12] Note that in the following transducer program we use an $auxiliary$ relation $P^{(3)}$