TLP **18** (5–6): 874–927, 2018. © Cambridge University Press 2018
doi:10.1017/S147106841800042X

A datalog-based computational model for coordination-free, data-parallel systems

MATTEO INTERLANDI*
Microsoft
(*e-mail:* mainterl@microsoft.com)

LETIZIA TANCA
Politecnico di Milano
(*e-mail:* letizia.tanca@polimi.it)

submitted 1 April 2017; revised 16 July 2018; accepted 23 July 2018

Abstract

Cloud computing refers to maximizing efficiency by sharing computational and storage resources, while *data-parallel systems* exploit the resources available in the cloud to perform parallel transformations over large amounts of data. In the same line, considerable emphasis has been recently given to two apparently disjoint research topics: *data-parallel*, and *eventually consistent, distributed* systems. *Declarative networking* has been recently proposed to ease the task of programming in the cloud, by allowing the programmer to express only the desired result and leave the implementation details to the responsibility of the run-time system. In this context, we deem it appropriate to propose a study on a *logic-programming-based computational model* for eventually consistent, data-parallel systems, the keystone of which is provided by the recent finding that the class of programs that can be computed in an eventually consistent, coordination-free way is that of *monotonic programs*. This principle is called Consistency and Logical Monotonicity (CALM) and has been proven by Ameloot *et al.* for distributed, asynchronous settings. We advocate that CALM should be employed as a basic theoretical tool also for data-parallel systems, wherein computation usually proceeds synchronously in rounds and where communication is assumed to be reliable. We deem this problem relevant and interesting, especially for what concerns *parallel dataflow optimizations*. Nowadays, we are in fact witnessing an increasing concern about understanding which properties distinguish synchronous from asynchronous parallel processing, and when the latter can replace the former. It is general opinion that coordination-freedom can be seen as a major discriminant factor. In this work, we make the case that the current form of CALM does not hold in general for data-parallel systems, and show how, using novel techniques, the satisfiability of the CALM principle can still be obtained although just for the subclass of programs called *connected monotonic queries*. We complete the study with considerations on the relationships between our model and the one employed by Ameloot *et al.*, showing that our techniques subsume the latter when the synchronization constraints imposed on the system are loosened.

* Work partially done while at University of California, Los Angeles.

KEYWORDS: declarative networking, datalog, relational transducer, CALM conjecture, bulk synchronous parallel systems.

1 Introduction

Recent research has explored ways to exploit different levels of *consistency* in order to improve the performance of distributed systems w.r.t.specific tasks and network configurations while preserving correctness (Brewer 2000; DeCandia *et al.* 2007; Vogels 2009). A topic strictly related to consistency is *coordination*, usually informally interpreted as a mechanism to accomplish a distributed agreement on some system property (Fagin *et al.* 2003). Indeed, coordination can be used to enforce consistency when, in the natural execution of a system, the latter is not guaranteed.

In this paper, we set forth a logic-programming-based framework to express database queries and study some theoretical problems springing from the use of eventually consistent, coordination-free computation over *synchronous systems with reliable communication* (*rsync* in short). *Rsync* is a common setting in modern data-parallel frameworks such as MapReduce (Dean and Ghemawat 2008), Pregel (Malewicz *et al.* 2010), and Apache Spark (Zaharia *et al.* 2012), where computation is commonly performed in *rounds*, and each task is blocked and cannot start the new round until a *synchronization barrier* is reached, i.e., every other task has completed its local computation. Identifying under what circumstances eventually consistent, coordination-free computation can be performed over *rsync* systems would enable the introduction of novel execution plans, no longer restricted by predefined (synchronous) patterns. For example, coordination-free programs can be divided into independent sub-units that can be run concurrently: a property known as *decomposability* (Wolfson and Silberschatz 1988). While our recent work (Shkapsky *et al.* 2016) implements the generalized pivoting technique (Seib and Lausen 1991) by which a decomposable plan can be identified from simple syntactic analysis of the program(s), still no semantics study exists on the matter.

Our aim is therefore to understand *in what generic circumstances a synchronous "blocking" computation is actually required by the program semantics – and therefore must be strictly enforced by the system – and when, instead, an asynchronous execution can be performed as optimization.* Recently, the class of programs that can be computed in an eventually consistent, coordination-free way has been identified: *monotonic programs* (Hellerstein 2010); this property is called *CALM* (Consistency and Logical Monotonicity) and has been proven in Ameloot *et al.* (2013). While CALM was originally proposed to simplify the specification of distributed (asynchronous) data management systems, in this paper, we advocate that CALM should be employed as a basic theoretical tool also for the declarative specification of data-parallel (synchronous) systems. As a matter of fact, CALM permits to link a property of the execution (coordination-freedom) to a class of programs, i.e., monotonic queries. But to which extent does CALM hold over data-parallel systems? Surprisingly enough, with the communication model and the notion of coordination as defined in Ameloot *et al.* (2013), the CALM principle does not hold in general in

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

rsync settings, the main reason being that the proposed definition of coordination is too weak to capture the type of coordination "baked" into the synchronization barrier of *rsync* systems (cf. Example 5). In this paper, we first characterize such type of coordination, then we study *to which extent the "synchronization barrier" creates coordination*, and finally we devise additional forms of coordination patterns.

To reach our goal, we develop a new *generic parallel computation model*, leveraging previous works on logic-based *relational transducers* (Abiteboul *et al.* 2000) and *transducer networks* (Ameloot *et al.* 2013), and grounding *rsync* computation on the well-known *Bulk Synchronous Parallel (BSP)* model (Valiant 1990). With BSP, computation proceeds in a series of global rounds, each comprising three phases: (i) a *computation phase*, in which nodes concurrently perform local computations; (ii) a *communication phase*, in which data is exchanged among the nodes; and (iii) the synchronization barrier. Exploiting this new type of transducer network, equipped with a content-based addressing model, we then show that the CALM principle is in general satisfied for BSP-style systems *under a new definition of coordination-freedom*, although, surprisingly enough, just for a subclass of monotonic queries, i.e., the *connected monotonic queries* (cf. Definition 10). When defining coordination-freedom, we will take advantage of recent results describing how knowledge can be acquired in synchronous systems (Ben-Zvi and Moses 2014). As a final outcome, a series of coordination patterns – and related classes of characteristic queries – is identified, and we will discuss how these coordination patterns behave under BSP and weaker synchronous settings. As a corollary, we show that the new definition of coordination-freedom subsumes the one employed in Ameloot *et al.* (2013).

Contributions: Summarizing, the contributions of the paper are as follows:

(1) The only-if direction of the CALM principle (namely that only monotonic queries can be computed in a coordination-free way) is proven not to hold in general for *rsync* systems (cf. Example 5 in Section 3.6).
(2) A novel, logic-programming-based computational model is introduced that emulates common patterns found in modern data-parallel frameworks (Section 4).
(3) A new definition of coordination is proposed, leveraging recent results on knowledge acquisition in *rsync* systems (Section 5).
(4) Exploiting the new techniques, the CALM principle is proven to hold for *connected monotonic queries* in *rsync* systems with bounded delay and deterministic data delivery (Theorem 1 in Section 6).
(5) A complete taxonomy of queries is provided that permits the identification of different types of coordination patterns (Section 6).
(6) The definition of coordination previously introduced in Ameloot *et al.* (2013) is shown to collapse into the one we propose, when the synchronization constraints assumed on the system model are loosened (Section 6.3).

Applications: Data-parallel programs such as the one implemented on top of MapReduce and Apache Spark relies on the assumption that computation is executed in rounds. This assumption makes the implementation of distributed programs easier because hides to the developers details on coordination and on how tasks are executed by the systems. Nevertheless, coordination-free execution is shown to be faster when allowed. In fact, certain algorithms are shown to converge

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

faster when (a bounded amount of) asynchrony is permitted (Niu *et al.* 2011; Cui *et al.* 2014). In general, it is well known that coordination-free (asynchronous) computations are amenable to *pipelining*, i.e., one-record-at-a-time executions of sequences of computations requiring no intermediate materialization of data; overall pipelining is highly desirable in the Big Data context, where full materialization is often problematic because of the limits on the available main memory. Finally, as previously mentioned, coordination-free programs are decomposable (Wolfson and Silberschatz 1988).

Currently, all high-level data-parallel languages are compiled into synchronous (blocking) plans; for instance, both Hive (Thusoo *et al.* 2009) and Pig (Olston *et al.* 2008) sacrifice efficiency in order to fit query plans into rounds of MapReduce jobs. Similarly, Spark SQL statically splits programs into stages separated by *ad-hoc* coordination logic. Other more sophisticated systems such as Hyracks (Borkar *et al.* 2011) and Apache Flink (Alexandrov *et al.* 2014) do provide the ability to pipeline operators, but it is the programmer's task to manually select the proper strategy. To our knowledge, only BigDatalog (Shkapsky *et al.* 2016) and few other systems (Niu *et al.* 2011; Cui *et al.* 2014; Han and Daudjee 2015; Xie *et al.* 2015) have started to explore when asynchronous executions can be delivered as optimizations of parallel, synchronous programs, and these are mainly in the graph-processing and machine-learning domain.

Organization: The rest of the paper is organized as follows: Section 2 introduces some preliminary notation. Section 3 defines our model of synchronous and reliable parallel system, 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 the new definition of coordination. Finally, Section 6 discusses CALM under the new setting. The paper ends with a comparison with other work and concluding remarks. The extended version of this paper (Interlandi and Tanca 2017) contains some additional material: (i) result on the decidability of independent specifications; and (ii) a complete study on the expressive power of the bulk synchronous transducer network model.

2 Preliminaries

The ultimate goal of this paper is to understand to which extent the execution of high-level data-parallel SQL-like languages such as Hive (Thusoo *et al.* 2009), Pig (Olston *et al.* 2008), and Spark SQL (Armbrust *et al.* 2015) can be optimized by understanding coordination patterns. In this section, we therefore recall the basic notions of database theory whereby queries (expressed as logic programs) are evaluated in a bottom-up fashion. We also set forth our notation, which is close to that of Abiteboul *et al.* (1995) and Ameloot *et al.* (2013).

2.1 Basic database notation

We denote by $\mathscr{D}$ an arbitrary *database schema* composed by a non-empty set of *relation schemas* (or simply *relations*). In the following, we will use the notation

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

$R^{(a)}$ 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* $R(\bar{u})$ is an ordered a-tuple over R composed by constants only. A *relation instance* I_R is a set of facts defined over $R \in \mathscr{D}$, while a *database instance* $\mathbf{I}$ is the union $\bigcup_{R \in \mathscr{D}} I_R$. In general, we write $I_{\mathscr{D}'}$ to denote an instance over the relations $\mathscr{D}' \subseteq \mathscr{D}$. The set $adom(\mathbf{I}) \subseteq \mathbf{dom}$ of all constants appearing in a given database instance $\mathbf{I}$ is called *active domain* of $\mathbf{I}$, while $inst(\mathscr{D})$ denotes the set of all the possible database instances defined over $\mathscr{D}$. Given a database schema $\mathscr{D}$ and a relation $R \in \mathscr{D}$, a *query* q_R is a total function such that $q_R : inst(\mathscr{D}) \rightarrow inst(R)$ and $adom(q(\mathbf{I})) \subseteq adom(\mathbf{I})$. In practice, we will only consider *generic* queries, i.e., if p is a permutation of **dom**, and $\mathbf{I}$ an input instance, then $q(p(\mathbf{I})) = p(q(\mathbf{I}))$. Finally, we say that a query is *monotonic* when given two instances $\mathbf{I}$, $\mathbf{J}$, if $\mathbf{J} \subseteq \mathbf{I}$, then $q(\mathbf{J}) \subseteq q(\mathbf{I})$. Note that in the above definitions, we have considered queries with a single output relation; this is not a limitation since queries with multiple output relations can be expressed as collections thereof: Given an input and output schema $\mathscr{D}_{\text{in}}$ and $\mathscr{D}_{\text{out}}$, respectively, we will write $\mathscr{Q} = \{q_R \mid R \in \mathscr{D}_{\text{out}}\}$. In this paper, we will consider the following query languages all expressible using a rule-based formalism: Unions of conjunctive queries UCQ, first order queries FO, DATALOG, and DATALOG with negation DATALOG$^\neg$. Next, we briefly introduce the syntax of the above query languages.

2.2 Query languages

Let **var** be an infinite set of *variables* ranging over the elements of **dom**. Given a relation R, an *atom* $R(\bar{u})$ is a tuple in which both constants and variables are permitted as terms. A *literal* is an atom – in this case, we refer to it as *positive* – or the negation of an atom.

A *conjunctive query with negation*, is an expression in the form of

$$H(\bar{w}) \leftarrow B_1(\bar{u}_1), \ldots, B_n(\bar{u}_n), \neg C_1(\bar{v}_1), \ldots, \neg C_m(\bar{v}_m), \tag{1}$$

where $H(\bar{w})$, $B_i(\bar{u}_i)$, and $C_j(\bar{v}_j)$ are atoms. As usual, $H(\bar{w})$ is referred to as the *head*, and $B_1(\bar{u}_1), \ldots, \neg C_m(\bar{v}_m)$ as the *body*. If $m = 0$, the rule is called *positive* while if $m = n = 0$ the rule is expressing a fact. For simplicity, in this paper, we assume each query to be *safe*, i.e., every variable, occurring either in a rule head or in a negative literal, appears in at least one positive literal of the rule body.

A UCQ query is a union of positive conjunctive queries, represented as a set of positive rules. In a DATALOG query, all predicates appearing in the body of a rule must be positive, and can also be used in the heads of rules to eventually produce *recursive* computation. A DATALOG$^\neg$ query is a set of safe rules where both recursion and negation are allowed. For DATALOG$^\neg$, we will assume the *stratified semantics*[1]. Finally, since non-recursive DATALOG$^\neg$ queries are equivalent to first-order logic (Abiteboul *et al.* 1995), we will use FO to denote such class of queries. In

[1] A DATALOG$^\neg$ program is said stratifiable if it can be partitioned into sub-programs (i.e., *strata*), each defining one or more negated predicates, and where no cycle of recursion contains a negated predicate (Abiteboul *et al.* 1995). According to the stratified semantics, these sub-program are then evaluated in order, following the dependencies among the negated predicates: Initially, the

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

this paper, we will only consider languages belonging to the above introduced set, with DATALOG$^{\neg}$ being the most expressive. Therefore, for each language $\mathscr{L}$, unless otherwise specified, we will assume $\mathscr{L} \subseteq$ DATALOG$^{\neg}$.

We will sometimes employ the function *sch* to return from a query its schema, i.e., $\mathscr{D} = sch(\mathscr{Q})$. The *intensional* (*idb*) part of the database schema is the subset of the database schema containing all the relations that appear in at least one non-fact rule head, while we refer to all the other relations in $sch(\mathscr{Q}) \setminus idb$ as *extensional* (*edb*).

2.3 Distributed systems

We define a *distributed system* as a *fully connected* graph of communicating nodes $N = \{1, \ldots, n\}$. We assign to each node i a *node configuration* denoted by the pair (N, i). We will in general assume all nodes to share the same *global* database schema. We will use the notation I_R^i to denote a *local instance* for node i over a relation R, while a *global instance* over R is defined as $I_R = \bigcup_{i \in N} I_R^i$. Given an initial database instance $\mathbf{I} \in inst(\mathscr{D}')$ defined over a subset of the global schema $\mathscr{D}$, we assume that a *distribution function* D exists mapping each node i to a (potentially overlapping) portion of the initial instance, this is $D : inst(\mathscr{D}') \times N \to inst(\mathscr{D}')$. For correctness, we assume that D is such that each fact composing the input instance is mapped to at least one node, i.e., $\bigcup_{i \in N} D(\mathbf{I}, i) = \mathbf{I}$. Finally, a *network configuration* is identified with the pair (N, D).

3 Computation in rsync

Query computability is usually defined using the classical model of computation: the Turing machine. However, in this paper, we are interested in the meaning of *computing a query in parallel settings*. To this end, in the following, we introduce a novel kind of *transducer network* (Ameloot *et al.* 2013), where computation is *synchronous* and communication is *reliable*, thus obtaining an abstract computational model for *distributed data-parallel systems*. As a first step, next we describe how *relational transducers* (Abiteboul *et al.* 2000; Ameloot *et al.* 2013) (hereafter, simply transducer) can be used to model local computations.

3.1 Relational transducers

We employ transducers as an abstraction modeling the behavior of each single computing node composing a computer cluster: This abstract computational model permits us to make our results as general as possible without having to rely on a specific framework, since transducers and *transducer networks* (introduced in the next section) can be easily used to describe any modern data-parallel system.

We consider each node to be equipped with an immutable *database*, and a *memory*-store used to maintain useful data between consecutive computation steps.

sub-programs having no dependency on negated predicates are fired, followed then by the strata depending on those that have just been executed, and so forth.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

In addition, a node can produce an *output* for the user and can also *communicate* some data with other nodes (data communication will be clarified in Section 3.2 with the concept of transducer network). An internal *time*, and *system* data are kept mainly for configuration purposes. Every node executes a *program* that translates a set of input instances (from the database, the memory and the communication channel), to a new set of instances that are either saved to memory, or directly output to the user, or addressed to other nodes. Programs are expressed in one of the languages of Section 2.1.

Formally, each node is modeled as a transducer $\mathscr{T}$ defined by the pair $(\mathscr{P}, \Upsilon)$ where $\mathscr{P}$ and Υ, respectively, denote the *transducer program* and the *transducer schema*. A transducer schema is a 6-tuple (Υ_{db}, Υ_{mem}, Υ_{com}, Υ_{out}, Υ_{time}, and Υ_{sys}) of disjoint relational schemas, respectively, called *database*, *memory*, *communication*, *output*, *time*, and *system* schemas. As default, we consider Υ_{sys} to contain two unary relations `Id`, `All`, while Υ_{time} includes just the unary relation `Time`, employed to store the current transducer local *clock* value[2]. A *transducer local state* over the schema Υ is then an instance I over $\Upsilon_{db} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{sys}}$. The transducer program $\mathscr{P}$ is composed by a collection of *insertion*, *deletion*, *output*, and *send* queries $Q_{\text{ins}} = \{q_R^{\text{ins}} | R \in \Upsilon_{\text{mem}}\}$, $Q_{\text{del}} = \{q_R^{\text{del}} | R \in \Upsilon_{\text{mem}}\}$, $Q_{\text{out}} = \{q_R^{\text{out}} | R \in \Upsilon_{\text{out}}\}$, and $Q_{\text{snd}} = \{q_R^{\text{snd}} | R \in \Upsilon_{\text{com}}\}$, all taking as input an instance over the schema Υ.

Starting from a relational transducer $\mathscr{T} = (\mathscr{P}, \Upsilon)$ and a node configuration (N, i), we can construct a *configured transducer*, denoted by $\mathscr{T}_N^i$, by setting $I_{Id} = \{\texttt{Id(i)}\}$ and $I_{All} = \{\texttt{All(j)} | \texttt{j} \in N\}$. Given a configured transducer and an instance $\mathbf{I}$ defined over Υ_{db}, we can create a *transducer initial local state* by setting $I_{db} = \mathbf{I}$. This basically models the starting status of a computing node, before the actual program execution starts: A node has received a program and a read-only instance (e.g., stored in a distributed file system such as HDFS where data is immutable) over which the computation must be performed, and has global knowledge of the other nodes composing the network. Indeed, this is exactly how working nodes are set up, for instance, in MapReduce or Spark.

Now, given a configured transducer $\mathscr{T}_N^i$, let I_{rcv}, J_{snd} denote two instances over Υ_{com} – and hence disjoint from I – with the former identifying a set of facts that have been previously sent to $\mathscr{T}_N^i$. If I is a local state, a *transducer transition*, denoted by $I, I_{rcv} \Rightarrow J, J_{\text{snd}}$ is such that J is the updated local state, while J_{snd} contains a set of facts that must be addressed to other transducers. The semantics for updates leaves the *database* and the *system* instances unchanged, while the facts produced by the insertion query Q_{ins} are inserted into the *memory* relations and all the facts returned by the deletion query Q_{del} are removed from them. 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 derived by the query Q_{out} are output. As regards J_{snd}, this is the set of facts returned by query Q_{snd} and sent by the transducer toward the other nodes. We assume that, once sent or output, *facts cannot be retracted*.

[2] The semantics of Υ_{time} will become clearer in Section 3.3 when we will describe the synchronous model.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

If $I' = I \cup I_{\text{rcv}}$, a transducer transition $I, I_{\text{rcv}} \Rightarrow J, J_{\text{snd}}$ is formally defined by the laws as follows:

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

Finally, note that transitions are deterministic, i.e., if $I, I_{\text{rcv}} \Rightarrow J$, J_{snd}, and $I, I_{\text{rcv}} \Rightarrow J'$, J'_{snd}, then $J = J'$ and $J_{\text{snd}} = J'_{\text{snd}}$.

Many different versions of transducers can be obtained by constraining the type of queries or the transducer schema. A transducer is *oblivious* if its queries do not use any *system* and *time* relations. Intuitively, this means that each query is unaware of the configuration, because independent of (i) the node it is running on, (ii) the other nodes in the network, and (iii) the time point at which the computation is. A transducer is called *monotonic* if all its queries are monotonic. Finally, we say that a transducer is *inflationary* if *memory* facts are never deleted – i.e., Q_{del} is empty.

Remark: The relational transducer model we have just defined is general, however it can be instantiated using a specific query language $\mathscr{L}$. We shall then write $\mathscr{L}$-transducer to denote that the program is actually implemented in $\mathscr{L}$.

Example 1

A first example of single-node relational transducer is the UCQ-transducer $\mathscr{T}$ below, computing an equi-join[3] between relations R and T[4].

Schema: $\Upsilon_{db} = \{R^{(2)}, T^{(2)}\}, \Upsilon_{\text{mem}} = \emptyset, \Upsilon_{\text{com}} = \emptyset, \Upsilon_{\text{out}} = \{Q^{(3)}\}$

Program: $Q_{\text{out}}(u, v, w) \leftarrow R(u, v), T(v, w).$

Let $\mathscr{T}^{i}_{N}$ be a configured version of $\mathscr{T}$, and $\mathbf{I}$ an initial instance over which we want to compute the join. Then, let $I_{db} = \mathbf{I}$. A transition for $\mathscr{T}^{i}_{N}$ is defined by setting $I = I_{db} \cup I_{\text{sys}}$, $I_{\text{rcv}} = J_{\text{snd}} = \emptyset$ (no communication query exists), and $J = I_{db} \cup I_{\text{out}} \cup I_{\text{sys}}$, where I_{out} is the result of the query on Q, i.e., the join between R and T.

Remark: Note that, to simplify the notation, in the example above, we omitted the schemas Υ_{sys} and Υ_{time} because they are always the same; for the same reason, henceforth we will also omit all the empty schemas, as in the case of Υ_{mem} and Υ_{com} in the example.

[3] The readers who are not familiar with database notation should consider that an equi-join operation between two relations is specified by sharing one or more variables. For example, in our case, the variable v, shared by R and T, indicates that the Q relation is obtained by imposing that the second term of R be equal to the first term of T.

[4] Note that, throughout the paper, we add a subscript to the relations in the rule heads to denote to which query – among the queries $Q_{\text{ins}}, Q_{\text{del}}, Q_{\text{out}}$, and Q_{snd} of the transducer program – the rule belongs.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

3.2 Transducer networks

We have already defined how local computations can be expressed by introducing the notion of relational transducer. In this section, we will model the behavior of a networked set of computing nodes by means of *specifications*. A *transducer network specification* (henceforth, simply *transducer network*, or *specification*) $\mathcal{N}$ is a tuple $(\mathcal{T}, \mathcal{T}^e, \gamma)$, where $\mathcal{T} = (\mathcal{P}, \Upsilon)$ is a transducer, $\mathcal{T}^e$ is a transducer defining the *environment*, and $\gamma : N \rightarrow inst(\Upsilon_{\text{com}})$ is a *communication* function mapping each node to a set of received facts. Transducer networks are defined such that all the nodes employ the same transducer $\mathcal{T}$, while the only thing that can be different from node to node is their state. Such an abstraction is thus appropriate for modeling data-parallel computation, where each node applies the same set of transformations in parallel over a portion of the initial data[5].

For the moment we will consider two types of specifications: *broadcasting* and *communication-free*. The former are specifications in which the communication function γ is such that every fact emitted by a transducer is sent to all the other transducers composing the network. In the latter, instead, every fact is delivered just locally to the sending node. In the remainder of this section, we will assume the network to be broadcasting.

The environment $\mathcal{T}^e$ is a "special" relational transducer. To give an intuition, following a common practice of multi-agent systems (Fagin *et al.* 2003), we use the environment for modeling all the non-functional concerns related to the system; in our specific case, data communication and synchronization. More precisely, we define $\mathcal{T}^e = (\mathcal{P}^e, \Upsilon^e)$ as a transducer, where Υ^e is composed only by the *memory* and *communication* relation schemas, where $\Upsilon^e_{\text{com}} = \Upsilon_{\text{com}}$ and Υ^e_{mem} is the primed copy of Υ^e_{com} (i.e., $\forall R \in \Upsilon^e_{\text{com}}, \exists R'$ s.t. $R' \in \Upsilon^e_{\text{mem}}$). The transducer program $\mathcal{P}^e$ contains, for each $R \in \Upsilon_{\text{com}}$, a set of queries of the form:

$$\begin{aligned} R'_{\text{ins}}(\bar{u}) &\leftarrow R(\bar{u}), \\ R_{\text{snd}}(\bar{u}) &\leftarrow R(\bar{u}), \end{aligned} \tag{2}$$

the first used to store into *memory* a copy of each tuple sent by each node of the network; the second to emit every received fact. The use of $\mathcal{T}^e$ will be further clarified in Section 3.3 when we introduce the operational semantics of transducer networks.

Given a network configuration (N, D), we denote by $\mathcal{N}_{N,D}$ a *configured transducer network*, i.e., a specification where all the transducers have been configured, and where each node i holds a database assigned according to the distribution function D. When $|N| = 1$ D returns the full instance, we call this the *trivial configuration*[6]. Thus, a configured transducer network can be used to model a cluster with a parallel system set up, and that is ready to run the user-submitted program; the trivial configuration represents a program that is run in local mode.

[5] Homogeneous transducers model what is known in the parallel computing world as Single Program, Multiple Data (SPMD) computations.

[6] Note that this follows from the previously introduced assumption on D that no facts from the database instance are ignored.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

A *transducer network global state* is a tuple $(I^e, I^1, \ldots, I^n)$ where, for each $j \in N \cup \{e\}$, the jth element is the related relational transducer state I^j. The definitions of oblivious, monotonic, and inflationary transducers provided at the end of Section 3.1 are naturally generalized over transducer networks. Now, given a specification $\mathscr{N}$, many possible executions may exist, each of them representing one possible evolution of the global state. We describe how network global states may change over time through the notion of *run* ρ, which binds *logical time* values to global states[7]. Then, if $\rho(t) = (I^e, I^1, \ldots, I^n)$ is the network global state at time t, a *point* (ρ^i, t) is the transducer state of node $i \in N$. We assume that the initial global state $\rho(0)$ is such that (i) the *database* local to each node contains the related partition of the initial instance; (ii) the local *system* relations are properly initialized with the node identifier and with the information about the other nodes; and (iii) all the other relations are empty.

Recall that 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 $\mathscr{N}_{N,D}$ and an initial instance $\mathbf{I} \in \text{inst}(\Upsilon_{\text{db}})$, we define a *system* $\mathscr{S}_{\mathscr{N}}(N, D, \mathbf{I})$ as a set of runs, where N, D, and $\mathbf{I}$ are the parameters shaping the system. Given a system $\mathscr{S}_{\mathscr{N}}(N, D, \mathbf{I})$, if its settings are irrelevant or clear from the domain, we will often denote it simply by $\mathscr{S}$.

In the following, we will also be interested in investigating *classes* of systems, i.e., sets 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 $\mathscr{N}$, by adding an instance and a *partial configuration*: i.e., a configuration having some unfixed parameter. Intuitively, if all the parameters are bound, we obtain a specific system $\mathscr{S}_{\mathscr{N}}(N, D, \mathbf{I})$. Partial configurations and classes of systems are important because, in the next sections, we will study with particular attention which (instantiated) specifications are able to obtain a unique final outcome, *independently of the provided configuration.*

3.3 Synchronous and reliable systems

We are mainly interested in synchronous systems with reliable communication. Informally, a distributed system is synchronous when all processing node's clocks run at the same rate and both the difference between two nodes clocks and the latency of data communication is bounded. One can construct a synchronous system by providing as input to each node an external reference *global clock*, and assuming that difference between the global clock received as input by each node is bounded [8]. A similar bound on data communication also have to exists, otherwise nodes will

[7] In this paper, we will consider logical time and *physical time* to be two different entities: the former is used to reason about the computation progress of a distributed system; the second can be thought as the time in seconds returned by a local call to the operating system.

[8] In fact, synchronization cannot be achieved if by the time the global clock reaches a node, the reference clock has changed significantly.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

be unable to reason about distributed property of the system. The next definition summarizes the above considerations.

Definition 1
A *synchronous system* $\mathscr{S}^{sync}$ 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.
S3 Emitted tuples arrive at destination at most after a certain bounded physical time Δ.

In our framework, the first property can be expressed by linking the time value stored in the `Time` relation of each node with the external logical time used to reason about system runs. This is accomplished by defining a *timed local transition* $I, I_{\text{rcv}} \overset{t}{\Rightarrow} J, J_{\text{snd}}$ as a local transition where, at each time instant t, $I_{\text{time}} = \{\texttt{Time}(t)\}$. In this enriched setting, we have that each transducer accepts as input also the clock value, and the environment can be employed to directly provide the clock driving the computation for all the transducers[9]. Under this perspective, each timed local transition is basically labeled with the time value in which it is performed.

The second property of Definition 1 can be added to our framework by assuming that programs proceed in *rounds*, and that each round, operationally speaking, lasts enough to permit the computation at each node to reach the fixpoint[10]. In the following, *w.l.o.g.* we will use the round number to refer to the time of the clock stored in the `Time` relation. Finally, in order to express the third property, we assume that emitted facts are first buffered locally, and then, once the node has completed its local transition, delivered in batch to destination.

The above properties imply that a remote tuple is either received after a bounded amount of time or never received. Recall that initially, however, we have required each system to be reliable, i.e., all emitted tuples arrive at destination:

Definition 2
To be reliable, a synchronous system must satisfy properties **S1**–**S3** along with the following additional conditions:

R1 In every run, for every received fact for node i at round t', there exists a node j and a round t s.t. $t < t'$ and a send query derived the same fact on node j in round t.
R2 In every run, if a fact has been emitted by a node i at round t, there exists a time t' s. t. $t < t'$ and the same fact belongs to the input instance state of a node j at round t'.

[9] We assume that the environment is the only transducer allowed to modify the time-related instances, which are then sent to the rest of the network to achieve synchronization.
[10] Note that, because of the considered languages, we are assured that local transitions always terminate in at most PTIME.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

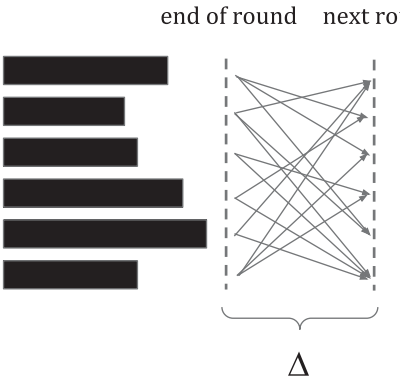

Fig. 1. *bsp* system computation model. Each node's computation is independent but bounded by the global end of the round (condition **S2**). Communication is reliable and takes exactly Δ time (conditions **S3** and **S3′**). The next round starts when Δ time since the end of the previous round has elapsed (conditions **S1** and **S3′**).

Informally, properties **R1** and **R2** specify that if an emitted fact exists in an instance at a given round, then it has been generated by a send query in a previous round, and vice-versa, if a send query derives a new fact, then that fact must appear in a successive round in the local state of a node. We denote by $\mathscr{S}^{\text{rsync}}$ the systems satisfying conditions **S1**–**S3** and **R1**–**R2**.

To further simplify the model, we can add to **S3** an extra condition forcing emitted tuples not only to be eventually received after at most Δ physical time, but to be delivered *exactly* after Δ:

S3′ Every tuple sent at round t is delivered exactly after Δ physical time.

We name this condition *deterministic delivery*: Without **S3′**, tuples may be non-deterministically delivered inside the bounded range defined by Δ and, as we will see in Section 6.2, this situation creates different coordination patterns. We can then assume that, between the end of one round and the start of the consecutive one, precisely Δ physical time elapses, so that all emitted tuples are received at the start of the new round. In other words, we can safely shift the beginning of each new round $t+1$ so that every tuple emitted at round t is precisely delivered at the start of the new round. We will use the signature $\mathscr{S}^{\text{bsp}}$ to denote a synchronous and reliable system with deterministic delivery. Figure 1 gives a pictorial representation of *bsp* systems. Note that this type of systems simulate how real-world BSP frameworks behave.

Next, we will show how the properties for *bsp* systems are enforced during global transitions.

3.4 Global transitions

Given a transducer network $(\mathscr{T}, \mathscr{T}^e, \gamma)$ and two global states $\mathbf{F} = (I^e, I^1, \ldots, I^n)$, $\mathbf{G} = (J^e, J^1, \ldots, J^n)$, let t be the clock value and σ be a *state* function mapping each node i and global state $\mathbf{F}$ to the corresponding local state I^i; i.e., $\forall i \in N, I^i = \sigma(\mathbf{F})(i)$.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Moreover, let $I^i_{\text{rcv}} = \gamma(i)$, and $I^i_{\text{db}} = D(\mathbf{I})(i)$. A *global transition* for a *bsp* system, denoted by $\mathbf{F} \Rightarrow \mathbf{G}$, is such that the following conditions hold:

- $(I^i, I^i_{\text{rcv}} \overset{t}{\Rightarrow} J^i, J^i_{\text{snd}})$ is a timed local transition for transducer $\mathscr{T}^i_N$.
- $(I^e, I^e_{\text{rcv}} \Rightarrow J^e, J^e_{\text{snd}})$ is a local transition for the environment, where $I^e_{\text{rcv}} = \bigcup_{i \in N} J^i_{\text{snd}}$.

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. In addition, in order to satisfy property **S3′**, we assume that a global transition can start only when 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 *bsp* system $\mathscr{S}^{\text{bsp}}$ is defined by *just one run*. We refer to the specifications defining *bsp* systems as *synchronous*.

3.5 Query computability

Given a run ρ describing the execution of a synchronous transducer network, we use the notation $out(t)$ for the set of facts output by all nodes at time t, i.e., $out(t) = \bigcup_{i \in N} I^i_{\text{out}}$ such that $I^i_{\text{out}} \in (\rho^i, t)$. This definition models how parallel data-processing frameworks work in practice: The output remains distributed on each node composing a cluster and can be eventually collected by invoking a proper function, or written to a distributed file system (e.g., HDFS). Now, let us assume that for a synchronous transducer network a time value t' exists such that $\forall t'' > t', out(t') = out(t'')$; that is, a quiescence state is reachable so that the output is stable and not changing any more. We define the output for a synchronous network to be the output up to network quiescence, denoted by $out(*)$, where we use $*$ to denote the time value in which the quiescence state is reached. In practice, a transducer network initial state identifies also its output. This is because the system $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}(N, D, \mathbf{I})$ is constituted by one and only one run, therefore, given an initial instance and a configuration, the output is uniquely determined. As a signature, we use $\mathscr{N}_{N,D}(\mathbf{I})$ to denote the output of the transducer network $\mathscr{N}_{N,D}$ on input database instance $\mathbf{I}$, and over a *bsp* system.

We are now able to state what we mean for a query to be computable by a transducer network $\mathscr{N}$:

Definition 3

Given an input and an output schema, respectively, $\mathscr{D}_{\text{in}}$ and $\mathscr{D}_{\text{out}}$, a total mapping $\mathscr{Q} : \text{inst}(\mathscr{D}_{\text{in}}) \rightarrow \text{inst}(\mathscr{D}_{\text{out}})$ is *computable by a synchronous transducer network* if a configured transducer network $\mathscr{N}_{N,D}$ exists such that $\mathscr{D}_{\text{in}} = \Upsilon_{db}$, $\mathscr{D}_{\text{out}} = \Upsilon_{\text{out}}$ and $\mathscr{N}_{N,D}(\mathbf{I}) = \mathscr{Q}(\mathbf{I})$, for every initial database instance $\mathbf{I}$ over $\mathscr{D}_{\text{in}}$.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Because we assumed generic queries as building blocks of transducers, we have that:

Proposition 1
The function $\mathscr{N}_{N,D}$ is generic for each oblivious synchronous transducer network.

Some specification might have the property that a unique final result is obtained independently of the chosen configuration; on the other hand, not all specifications have this property. Note that this property is a common requirement in parallel systems: the same job must return consistent results, whichever the cluster size and partitioning scheme.

Definition 4
Given a specification $\mathscr{N}$, an input instance $\mathbf{I} \in \text{inst}(\Upsilon_{\text{db}})$, and a partial configuration ψ, we say that the class $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}(\psi, \mathbf{I})$ is *convergent* if for all pairs $\rho, \rho' \in \mathscr{S}^{\text{bsp}}_{\mathscr{N}}(\psi, \mathbf{I})$, the respective final outputs out(∗), out′(∗) coincide, i.e., out(∗) = out′(∗).

Informally, a class of systems is convergent if, at the quiescence state, all its runs have the same output. Note that – in order for the convergence property to hold – each pair of runs are required to agree just on the initial instance $\mathbf{I}$ and on the final output state, and not necessarily on the entire execution. Again, this means that, whichever configuration we select, we are assured that the same final outcome will be eventually returned. We will then say that a specification $\mathscr{N}$ is *network-independent* if, once fixed a distribution function D, the class $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}(D, \mathbf{I})$ is convergent for all possible $\mathbf{I} \in \text{inst}(\Upsilon_{\text{db}})$, or, in other words, $\mathscr{N}_{N,D}$ computes the same query $\mathscr{Q}$ for all networks N and instances $\mathbf{I}$[11]. A similar definition applies for *distribution-independent* specifications. Finally, if a specification $\mathscr{N}$ is network-distribution-independent, the class $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}$ is convergent, i.e., for any instance $\mathbf{I}$, all the possible runs in $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}(\mathbf{I})$ compute the same query result $\mathscr{Q}(\mathbf{I})$. This is because, whichever configuration is selected, $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}(\mathbf{I})$ has a unique output. In this case, $\mathscr{Q}$ is said to be *distributively computable*, while $\mathscr{N}$ is said to be *independent* (or *convergent*). Note that similar definitions have been used also in Ameloot *et al.* (2013), where it was also shown that independence is an undecidable property of specifications[12].

Example 2
Assume we want to compute a distributed version of the program of Example 1. This can be implemented using a broadcasting and inflationary synchronous transducer network in which every node emits one of the two relations, let us say T, and then joins R with the facts over T received from the other nodes. Note that the sent facts will be used just starting from the successive round. This program will then employ

[11] Recall that we only consider fully connected networks, i.e., networks are only distinguished by their cardinality. This too is a common assumption in modern data-parallel distributed systems.

[12] Note that we could have added the initial round number to the network configuration parameters. Instead we chose to fix to 0 the initial round value not to make our model too complex. Because of this, we implicitly assume each query to be *time-independent*: the value stored in the `Time` relation does not influence the result of the query. A (conservative) syntactical condition to achieve time-independence is to not have the `Time` relation in any query-rule.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

two rounds to compute the distributed join. UCQ is again expressive enough. The transducer can be written as follows:

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

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 $\mathscr{L}$ be a language containing UCQ and such that $\mathscr{L} \subseteq$DATALOG$^{\neg}$. Every query expressible in $\mathscr{L}$ can be distributively computed in two rounds by a broadcasting, inflationary, and oblivious $\mathscr{L}$-transducer network.

Proof
This is an adaptation to our context of Theorems 4.9–4.10 of Ameloot *et al.* (2013). Let $\mathscr{Q}$ be a query expressed in $\mathscr{L}$ having input schema $\mathscr{D}_{\text{in}}$ and output schema $\mathscr{D}_{\text{out}}$. We have two cases: (i) $\mathscr{Q}$ is monotonic or (ii) $\mathscr{Q}$ is non-monotonic. In case (i), we can program a transducer $\mathscr{T}$ 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. $\mathscr{Q}$ is evaluated over the union of the local input with the received facts.

Formally, the transducer schema will have $\Upsilon_{\text{db}} = \mathscr{D}_{\text{in}}$, $\Upsilon_{\text{com}} = \{R'^{(a)} \mid R^{(a)} \in \Upsilon_{\text{db}}\}$, $\Upsilon_{\text{out}} = \mathscr{D}_{\text{out}}$, while the *system* and *time* schemas are as usual. Denote with $\mathscr{Q}'$ the version of the query $\mathscr{Q}$ where all the *edb* relations are primed. The transducer program $\mathscr{P}$ is composed by the queries Q_{snd} and Q_{out}, where Q_{snd} is composed by one rule in the form: $R'_{\text{snd}}(\bar{u}) \leftarrow R(\bar{u})$ for each $R \in \Upsilon_{\text{db}}$, while Q_{out} simply contains $\mathscr{Q}'$. The specification is monotonic and oblivious.

In case (ii), we follow the same approach as before, but this time the query $\mathscr{Q}$, being non-monotonic, cannot be applied immediately because wrong results could be derived. To avoid this, we use the transducer $\mathscr{T}$ of case (i), in which we add to Υ_{mem} the nullary relation `Ready`, and to Q_{ins} the query: $\texttt{Ready}_{\text{ins}}() \leftarrow \neg\texttt{Ready}()$. In addition, we modify the rules in $\mathscr{Q}'$ by adding the literal `Ready` to their body. In this way, the query $\mathscr{Q}$ will be evaluated just starting from the second round since at the first one `Ready` is false. We therefore reach our goal. □

Examples 3 and 4 below show two transducers, each computing a query of one of the two categories used in the proof of Lemma 1.

Example 3
Let $\mathscr{Q}$ be the following UCQ-query:

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

$\mathscr{Q}$ can be computed by the following UCQ-transducer:

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

$$\begin{aligned}
\text{Schema: } & \Upsilon_{db} = \{R^{(1)}, P^{(2)}\}, \Upsilon_{\text{snd}} = \{R'^{(1)}, (P'^{(2)}\}, \\
& \Upsilon_{\text{out}} = \{T^{(2)}\} \\
\text{Program: } & R'_{\text{snd}}(u) \leftarrow R(u). \\
& P'_{\text{snd}}(u,v) \leftarrow P(u,v). \\
& T_{\text{out}}(u,v) \leftarrow P'(u,v), R'(u).
\end{aligned}$$

Example 4
Let $\mathcal{Q}$ be the following (non-monotonic) FO-query:

$$\begin{aligned}
T(u,z) &\leftarrow R(u,v), P(v,z), \\
Q(u,z) &\leftarrow S(u,v), \neg T(v,z), P(w,z),
\end{aligned}$$

with $\mathscr{D}_{\text{in}} = \{R^{(2)}, P^{(2)}, S^{(2)}\}$ and $\mathscr{D}_{\text{out}} = Q^{(2)}$. An FO-transducer computing the same query is

$$\begin{aligned}
\text{Schema: } & \Upsilon_{\text{db}} = \{R^{(2)}, P^{(2)}, S^{(2)}\}, \Upsilon_{\text{mem}} = \{\texttt{Ready}^{(0)}\}, \\
& \Upsilon_{\text{snd}} = \{R'^{(2)}, P'^{(2)}, S'^{(2)}\}, \Upsilon_{\text{out}} = \{Q^{(2)}\} \\
\text{Program: } & R'_{\text{snd}}(u,v) \leftarrow R(u,v). \\
& P'_{\text{snd}}(u,v) \leftarrow P(u,v). \\
& S'_{\text{snd}}(u,v) \leftarrow S(u,v). \\
& \texttt{Ready}_{\text{ins}}() \leftarrow \neg \texttt{Ready}(). \\
& T(u,z) \leftarrow R'(u,v), P'(v,z), \texttt{Ready}(). \\
& Q(u,z)_{\text{out}} \leftarrow S'(u,v), \neg T(v,z), P'(w,z), \texttt{Ready}().
\end{aligned}$$

Lemma 1 permits us to draw the following conclusion: Under the *bsp* 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 (Ameloot *et al.* 2013), we are guaranteed by the reliability of the communication and the synchronous assumption that, starting from the second round on, every node will compute the query over every emitted instance. Conversely, in the asynchronous case, as a result of the non-determinism of the communication, we are never guaranteed, without coordination, when every sent fact will be actually received. As a consequence, under this latter model, we do not know – without coordination – when negation can be safely applied, because it could be applied "too early", i.e., before all facts over the negated literal are received (or deduced).

3.6 The CALM conjecture

The *CALM conjecture* (Hellerstein 2010) specifies that the class of *monotonic programs* can be distributively computed in an eventually consistent, *coordination-free* way. CALM has been proven in this (revisited) form for asynchronous systems (Ameloot *et al.* 2013):

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Conjecture 1

A query can be distributively computed by a coordination-free transducer network if and only if it is monotonic.

Surprisingly enough, the only-if direction on the conjecture does not hold in *bsp* settings under the broadcasting communication model. Before showing this, we have to adapt the definition of *coordination-free* as expressed in Ameloot *et al.* (2013) to our synchronous model.

The concept of coordination suggests that all the nodes in a network need to exchange information and wait until an agreement is reached about a common property of interest. Following this intuition, Ameloot *et al.* established that a specification is coordination-free if communication is not strictly necessary to obtain a consistent final result. Put in a more formal context: A specification is coordination-free if (i) it is independent and (ii) a "perfect" distribution function exists such that communication is not required to achieve the final outcome. That is, the class generated from that specification admits a unique output, independently of the configuration. Hence, among all the possible distribution functions we can select one such that, if we turn communication off, the correct result is still returned.

Definition 5

Let $\mathscr{N}$ be an independent specification, and $\mathscr{F}$ its communication-free version. We say that $\mathscr{N}$ is *coordination-free* if $\forall \mathbf{I} \in \text{inst}(\Upsilon_{\text{db}})$ a non-trivial configuration (N, D) exists s.t., $\mathscr{S}^{\text{bsp}}_{\mathscr{F}}$ is convergent, where $\mathscr{S}^{\text{bsp}}_{\mathscr{F}}$ is constructed by adjoining to the convergent class $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}(\mathbf{I})$ the run defining $\mathscr{S}^{\text{bsp}}_{\mathscr{F}}(N, D, \mathbf{I})$.

In the following, we will say that runs such as $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}(N, D, \mathbf{I})$ and $\mathscr{S}^{\text{bsp}}_{\mathscr{F}}(N, D, \mathbf{I})$ are *eventually consistent* with each other. That is, their complete execution may differ, but they eventually converge to the same unique final output. To simplify the notation, we will sometimes directly say that $\mathscr{N}$ and $\mathscr{F}$ are eventually consistent.

It is now easy to see that with this definition there are non-monotonic queries that can be distributively computed by coordination-free specifications, as the next example shows.

Example 5

Let $\mathscr{Q}_{\text{emp}}$ be the "emptiness" query of Ameloot *et al.* (2013): Given a nullary database relation schema $R^{(0)}$ and a nullary output relation $T^{(0)}$, $\mathscr{Q}_{\text{emp}}$ outputs T iff I_R is empty. The query is non-monotonic: if I_R is initially empty, then T is produced, but if just one fact is added to R, T is not derived, i.e., I_T is now empty. A broadcasting FO-transducer network $\mathscr{N}$ can be easily generated to distributively compute $\mathscr{Q}_{\text{emp}}$: first, every node 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.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Schema: $\Upsilon_{db} = \{R^{(0)}\}, \Upsilon_{\text{mem}} = \{\texttt{Ready}^{(0)}\}, \Upsilon_{\text{com}} = \{S^{(0)}\}$,
$\Upsilon_{\text{out}} = \{T^{(0)}\}$.

Program: $S_{\text{snd}}() \leftarrow R()$.
$\texttt{Ready}_{\text{ins}}() \leftarrow \neg\texttt{Ready}()$.
$T_{\text{out}}() \leftarrow \neg S(), \texttt{Ready}()$.

Assume that (N, D) is a non-trivial configuration and let $\mathscr{F}$ be the communication-free version of $\mathscr{N}$ above. Clearly, whichever initial instance $\mathbf{I}$ we select, $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}(N, D, \mathbf{I})$ and $\mathscr{S}^{\text{bsp}}_{\mathscr{F}}(N, D, \mathbf{I})$ are eventually consistent when D installs $\mathbf{I}$ on every node.

Note that, in asynchronous settings, the same query requires coordination: Since emitted facts are non-deterministically received, the only way to compute the correct result is that every node coordinates with each other in order to understand if the input instance is *globally* empty.

The above example shows that the only-if direction of CALM, stating that only monotonic queries can be computed in a coordination-free way, *does not hold in general in synchronous settings* (Contribution 1). This result is indeed interesting although expected: When we move from the general asynchronous model to the *more restrictive,* bsp *setting*, we no longer have a complete understanding of which queries can be computed without coordination, and which ones, instead, do require coordination. It turns out that the communication model and the definition of coordination proposed in Ameloot *et al.* (2013) cannot capture a notion of coordination freedom appropriate also for synchronous systems. As the reader may have realized, this is due to the presence of (i) broadcasting communication and (ii) *bsp* system semantics: In broadcasting synchronous systems, the form of coordination defined by Ameloot *et al.* is already "baked" into the model because synchronization barriers *per se* provide nodes with the possibility of "indirectly deducing" the global status of the network. As a result, some of the queries that, in the asynchronous communication model, were not computable in a coordination-free way turn out to be so in synchronous systems. Arguably, one could conjecture that all computable queries are actually coordination-free computable (using again the notion introduced in Ameloot *et al.* (2013)) in synchronous systems.

In Section 5.2, we provide a new, less permissive definition of *coordination-freedom*, more appropriate for synchronous settings. Under this definition, *the discriminating condition for coordination freedom is not the absence of communication among nodes, but the more restrictive one that nodes do not need to acquire knowledge of a global property of the network to correctly compute a query*. We will then see that by weakening first (i) (Section 4) and then (ii) (Section 6), the coordination baked into the synchronization barrier results inappropriate to make specific classes of queries (described later) consistent, whereby additional forms of coordination are required. In Section 6, we will additionally show that indeed our definition of coordination subsumes the one previously introduced by Ameloot *et al.*: When we remove all the constraints imposed on the synchronous system, the two definitions fall together.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

The intuition is that, if a node does not know when all the other nodes are "done," conclusions (such as deducing that negation can be applied over a relation) might be applied "too early." That is, in order to apply certain specific types of deductions (which specific type will be made clear in the next sections), *common knowledge* (Fagin *et al.* 2003) is required. Acquiring common knowledge of a global property in asynchronous systems requires exchange of messages among all nodes, and this is why communication-freedom can be connected to coordination freedom in this type of systems (i.e., no coordination can be reached without communication). Conversely, using synchronous broadcasting specifications, common knowledge can be acquired by each node indirectly, even without having any fact being communicated.

4 Hashing transducer networks and parallel computation

In the previous section, we have seen that synchronization barriers, together with a broadcasting communication model, allow non-monotonic queries to be computed in a coordination-free way (cf. Example 5); more than that, broadcasting specifications do not appear to be really useful from a practical perspective. As a consequence, following other parallel programming models such as MapReduce and Spark, in this section, we are going to introduce *hashing transducers* (Contribution 2), i.e., relational transducers equipped with a *content-based communication model*. Under this new model, the node to which an emitted fact must be addressed is derived using a hash function applied to a subset of its terms called *keys*.

4.1 Hashing transducer networks

Let Υ be a transducer schema. For each relation $R^{(a)} \in \Upsilon_{\text{com}}$, we fix a subset of its terms as the key-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. As a notation, we associate to every transducer schema Υ a *key-set* $\mathscr{K}$ mapping every relation R for which a key is defined, to the related set of keys.

It is now appropriate to define how a hashing transducer $\mathscr{T}$ is represented. $\mathscr{T}$ is hashing if defined by a tuple $(\mathscr{P}, \Upsilon, \mathscr{K})$, where $\mathscr{P}$ is the transducer program, Υ the schema, and $\mathscr{K}$ a key-set. With each transducer network, we can now associate a *distributed hash mapping* $\mathscr{H}$ binding each *communication* fact with the non-empty set of nodes to which the fact belongs. Given a family H of unary hash functions $h : \mathbf{dom} \to N$ and a fact $R(\mathrm{u}_1, \ldots \mathrm{u}_a)$ over a relation $R^{(k,a)} \in \Upsilon_{\text{com}}$, $\mathscr{H}$ distributes $R(\mathrm{u}_1, \ldots \mathrm{u}_a)$ to the nodes in the following way:

$$\mathscr{H}(R(\mathrm{u}_1, \ldots \mathrm{u}_a)) = \begin{cases} N & \text{if } k = 0 \\ \bigcup_{i \in 1..k} \{h_i(\mathrm{u}_i)\} & \text{otherwise} \end{cases} \tag{3}$$

Informally, we employ hash functions to deterministically obtain the location(s) to which a fact belongs from its key-term values specified in $\mathscr{K}$, so that $\mathscr{H}$ maps each fact to the set of nodes to which it must be delivered. Two characteristics are noteworthy in our model. First, we allow a fact to be hashed to multiple target

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

nodes[13]. This multicasting model enables us to gracefully move from broadcasting to unicast situations, while studying how different classes of queries behave. Additionally, this allows to model both regular MapReduce-style shuffling, as well as Hypercube shuffling (Afrati and Ullman 2010) requiring records to be replicated over multiple nodes. Second, we consider a family of hash functions instead of a single function. In this way, we are able to express specific behaviors when needed; for instance, we can establish that facts containing a certain constant in certain cases are addressed to a predefined node, while, in general, they can be addressed to all nodes. To maintain our model simple, we assume the codomain of each hash family to coincide with N, i.e., $N = \bigcup_{h \in H} \bigcup_{c \in \mathbf{dom}} \{h(c)\}$.

Given a relation $R^{(k,a)}$, two key-settings are of special interest: (i) no key is set, and we write $k = 0$; and (ii) the set of keys is *maximal*, i.e., $k = a$. In the former case, we have that every tuple is addressed to all the nodes N. We then say that a send query is *broadcasting* if the head relation R is such that $k = 0$. Furthermore, in the case in which all the relations in the domain of $\mathscr{K}$ have $k = 0$, we say that $\mathscr{K}$ is *unrestricted*; it is *restricted* if, instead, for no relation $k = 0$. We can now base the definition of the communication function γ on $\mathscr{H}$ and we then call *hashing* this new type of synchronous specification. Unless stated otherwise, from now on, the term "specification" will actually denote a *hashing* specification (transducer network). The definition of communication-free can now be slightly revisited: a hashing specification $\mathscr{N}$ is communication-free also when, for all relations $R \in \Upsilon_{\text{com}}$, if $R(\bar{u})$ is a fact derived at node i by a sending query, then $\mathscr{H}(R(\bar{u})) = \{i\}$.

Example 6
This program is the hashed version of Example 2:

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

$$\begin{aligned} \text{Program: } & S_{\text{snd}}(u,v) \leftarrow R(u,v). \\ & U_{\text{snd}}(u,v) \leftarrow T(u,v). \\ & J_{\text{out}}(u,v,w) \leftarrow S(u,v), U(u,w). \end{aligned}$$

In this new guise, every tuple emitted over S and U is hashed on the first term, so that we are assured that at least a node exists to which each pair of joining tuples is issued. Note that such program models exactly how (reduce-side) joins are implemented in the MapReduce framework.

Shuffling transducers. Given a hashing transducer network, we can directly apply hashing functions to the *database* relations and, once all the initial tuples are hashed, the actual queries can be applied. We refer to such type of transducers as

[13] Note that this one-to-many distributed mapping is used here for coherence with the set oriented DATALOG semantics. One-to-one communication behavior can also be employed, for example, by using surrogate keys.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

shuffling. Intuitively, when $\mathscr{K}$ is unrestricted, the same behavior described in the proof of Lemma 1 under the broadcasting communication model is obtained.

More formally, given a query $\mathscr{Q} : \text{inst}(\mathscr{D}_{\text{in}}) \rightarrow \text{inst}(\mathscr{D}_{\text{out}})$ over input and output schema $\mathscr{D}_{\text{in}}$ and $\mathscr{D}_{\text{out}}$, respectively, a shuffle transducer network can be devised so that the transducer schema Υ has $\Upsilon_{\text{db}} = \mathscr{D}_{\text{in}}, \Upsilon_{\text{mem}} = \{\texttt{Ready}^{(0)}\}, \Upsilon_{\text{snd}} = \mathscr{D}'_{\text{in}}$, and $\Upsilon_{\text{out}} = \mathscr{D}_{\text{out}}$, where $\mathscr{D}'_{\text{in}}$ is composed by the primed version of the relations in $\mathscr{D}_{\text{in}}$; the transducer program $\mathscr{T}$ is instead formed by the queries $Q_{\text{ins}} = \{\texttt{Ready}_{\text{ins}}() \leftarrow \neg\texttt{Ready}().\}$, $Q_{\text{snd}} = \{R'_{\text{snd}}(\bar{u}) \leftarrow R(\bar{u}) | R \in \Upsilon_{\text{db}}\}$, and $Q_{\text{out}} = \mathscr{Q}'$, where $\mathscr{Q}'$ is generated from $\mathscr{Q}$ by priming all relations over the input schema $\mathscr{D}_{\text{in}}$ and where the predicate Ready() is adjoined to the query.

We will often use shuffling transducers in the proofs and examples we are going to introduce in the remainder of this section and in the following ones. But we first introduce some properties of shuffling specifications.

4.2 Properties of shuffling transducer networks

Safety and *liveness* are common properties used to describe specifications over distributed systems (Kindler 1994). 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 two properties can be, respectively, restated as "*no wrong fact will ever be derived*," and "*some fact will eventually be derived*."

First of all, note that, by the properties **S1**–**S3′** and **R1**–**R2** of *bsp* systems and by definition of hashing communication, every fact $P(\bar{u})$ over a relation $P \in \Upsilon_{\text{com}}$ derived by a send query in round t, will be in the instance I_P^i of node i at round $t+1$, for all $i \in \mathscr{H}(P(\bar{u}))$. Starting from the above consideration, we can define a query to be live if all the derived *communication* facts (globally) satisfying the body of a query are all co-located on at least one node. A specification is then live if all the queries satisfy the liveness property. More formally:

Definition 6

Let $\mathscr{T} = (\mathscr{P}, \Upsilon, \mathscr{K})$ be a shuffling transducer, and $\mathbf{I}$ an arbitrary instance over Υ_{db}. Assume that a query q_R in $\mathscr{P}$ exists such that q_R is satisfied in the trivial configuration of $\mathscr{T}$ with input $\mathbf{I}$. Let $\mathbf{q}$ be one of such instantiation of q_R, and denote with $\mathbf{b}$ the set of facts over the *communication* predicates in the body of $\mathbf{q}$. Then, q_R is *live* if for every instantiation $\mathbf{q}$ and (non-trivial) configuration we have that:

$$\left(\bigcap_{P(\bar{u}) \in \mathbf{b}} \mathscr{H}(P(\bar{u}))\right) \neq \emptyset \tag{4}$$

$\mathscr{T}$ is said to be live if the above property holds for all input instance $\mathbf{I}$ and queries in $\mathscr{P}$.

By showing that a query is live, we can state that a (shuffling) specification has at least the same opportunity to distributively derive a fact as the original query computed locally on a single node.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

On the other hand, a query is considered safe if every node evaluates all the negated literals on an instance containing all the facts, over that literal, available in the network:

Definition 7
Let $\mathscr{T} = (\mathscr{P}, \Upsilon, \mathscr{K})$ be a shuffling transducer, and N a set of nodes. We say that evaluating a query q_R in $\mathscr{P}$ is *safe* if, for every fact $P(\bar{u})$ over a relation $P \in \Upsilon_{\text{com}}$ appearing negated in the body of q_R, we have that

$$\forall i \in N, i \in \mathscr{K}(P(\bar{u})) \tag{5}$$

$\mathscr{T}$ is safe if every query in $\mathscr{P}$ can be safely evaluated on every input instance.

If a query is not safe, the correctness of a specification can be jeopardized, as shown next.

Example 7
Consider the following FO-query:

$$Q(v, w) \leftarrow R(u, v, w), \neg P(v, w)$$

with $\mathscr{D}_{\text{in}} = \{R^{(3)}, P^{(2)}\}$ and $\mathscr{D}_{\text{out}} = Q^{(2)}$. The following shuffling FO-transducer $\mathscr{N}$ implementing the same query is not safe:

Schema: $\Upsilon_{\text{db}} = \{R^{(3)}, P^{(2)}\}, \Upsilon_{\text{mem}} = \{\texttt{Ready}^{(0)}\},$
$\Upsilon_{\text{snd}} = \{S^{(1,3)}, T^{(1,2)}\}, \Upsilon_{\text{out}} = \{Q^{(2)}\}$

Program: $S_{\text{snd}}(u, v, w) \leftarrow R(u, v, w).$
$T_{\text{snd}}(u, v) \leftarrow P(u, v).$
$\texttt{Ready}_{\text{ins}}() \leftarrow \neg\texttt{Ready}().$
$Q(v, w)_{\text{out}} \leftarrow S(u, v, w), \neg T(v, w), \texttt{Ready}().$

Assume the following input instance $\mathbf{I} = \{\texttt{R(1,2,3)}, \texttt{P(2,3)}\}$. Clearly, the original query will have empty output over this input instance. However, let (N, D) be a non-trivial configuration, and H a hash family such that constants `1` and `2` are hashed to two different nodes. We then have that $\mathscr{N}$ outputs the fact `Q(2,3)` when using this configuration, i.e., $\mathscr{N}$ is not safe.

4.3 Parallel computation of queries

In our effort toward the creation of a general model for parallel computation, having modeled communication by means of hashing we are now able to create different computational strategies by simply customizing the adopted hash functions and relation keys. While we have already seen that $\mathscr{K}$ is embedded directly into the definition of a transducer, H can be added to the configuration parameters of transducers and specifications. Henceforth, for configured hashing transducers (resp. transducer networks), we will refer to transducers (networks) determined by configurations of the forms (N, i, H) (respectively (N, D, H)). The definition of

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

independent specification can now be extended accordingly. Thus, once a set of keys $\mathscr{K}$ has been fixed, a specification $\mathscr{N}$ is called *strategy independent* if, whatever H is chosen, $\mathscr{N}$ computes the same final result. Finally, a hashing specification is called *independent* if it is altogether network-distribution-strategy-independent. The definition of convergent class of systems (Definition 4) can naturally be generalized over (shuffling) hashing specifications.

In the next sections, we will explore in detail both the connections existing between class of queries and type of specifications, and which class is more expensive to compute with respect to others (e.g., which query can only be computed by an unrestricted specification).

In Section 3, we have seen that, if the transducer network is broadcasting, a large class of queries can be distributively computed. We say that a query $\mathscr{Q}$ is *parallelly computable* if a hashing specification exists such that all the possible runs in $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}(\mathbf{I})$ compute the same query $\mathscr{Q}(\mathbf{I})$, whichever initial instances $\mathbf{I}$ is given.

Proposition 2

Let $\mathscr{L}$ be a query language. Every query that is expressible in $\mathscr{L}$, and that can be distributively computed by a broadcasting $\mathscr{L}$-transducer network, can also be parallelly computed by a hashing $\mathscr{L}$-transducer network.

Proof

By definition, every hashing transducer with unrestricted $\mathscr{K}$ emulates a broadcasting one. □

It is now a straightforward exercise to show that the expressiveness result of Lemma 1 applies also to hashing specifications. *W.l.o.g.* we will hereafter call *broadcasting* every hashing transducer network where $\mathscr{K}$ is unrestricted.

Correct specifications. In Section 4.3, we have seen that the set of keys $\mathscr{K}$ can be used to parametrize a transducer. In this way, multiple specifications can be produced by selecting different sets of keys. Let $\mathscr{N}$ be the class of specifications that can be generated by changing $\mathscr{K}$ in a specification $\mathscr{N}$. Intuitively, a wrong selection of the keys can result in a wrong specification: Consider Example 6 and assume that we choose 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. Yet, we can define a subclass of $\mathscr{N}$ wherein each specification is consistent with the broadcasting version. By Proposition 2, in fact, a "correct" broadcasting specification always exists and it does not depend on the chosen set of keys: it simply broadcasts every emitted fact! Specifically, let $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}$ be the convergent class derived from the broadcasting specification $\mathscr{N}$. For every input instance $\mathbf{I}$, we can start to add to $\mathscr{S}^{\text{bsp}}_{\mathscr{N}}$ all the runs $\mathscr{S}^{\text{bsp}}_{\mathscr{N}'}(N, D, H, \mathbf{I})$ such that $\mathscr{N}' \in \mathscr{N}$ is a convergent specification, and $\mathscr{S}^{\text{bsp}}_{\mathscr{N}'}(N, D, H, \mathbf{I})$ and $\mathscr{S}^{\text{bsp}}_{\mathscr{N}''}(N, D, H, \mathbf{I})$ are eventually consistent. Denote with $\mathscr{S}^{\text{bsp}}_{\mathscr{C}\mathscr{N}}$ the class which is maximal with the above property, and where with $\mathscr{C}\mathscr{N} \subseteq \mathscr{N}$, we identify the *correct* class of specifications. We have the following:

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Definition 8
Let $\mathcal{N} = (\mathcal{T}, \mathcal{T}^e, \gamma)$ be a hashing transducer network with $\mathcal{T} = (\mathcal{P}, \Upsilon, \mathcal{K})$, and assume that a non-empty correct class of specifications $\mathscr{C}\mathcal{N}$ exists. A set of keys $\mathcal{K}$ (specification $\mathcal{N}$) is said to be *correct* if $\mathcal{N} \in \mathscr{C}\mathcal{N}$.

Note that the specifications in $\mathscr{C}\mathcal{N}$ are independent by construction.

4.4 Queries computable by a restricted specification

Among the parallelly computable queries, the most interesting ones from a parallel processing perspective are the ones for which a restricted specification (i.e., having a restricted set of keys), parallelly computing them, exists. We denote this kind of queries as *restricted*. Intuitively, restricted queries are the ones which can be parallelly computed without having to resort to broadcasting. Note that the reader should not be deluded into believing that every monotonic query is trivially a restricted one.

Example 8
Assume two relations $R^{(2)}$ and $T^{(1)}$, and the following query $\mathcal{Q}$ returning the full R-instance if T is non-empty.

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

The query is monotonic. Let $\mathcal{T}$ be the following broadcasting UCQ-transducer computing $\mathcal{Q}$.

Schema: $\Upsilon_{\text{db}} = \{R^{(2)}, T^{(1)}\}, \Upsilon_{\text{com}} = \{S^{(0,2)}, U^{(0,1)}\}, \Upsilon_{\text{out}} = \{Q^{(2)}\}$

Program: $S_{\text{snd}}(u,v) \leftarrow R(u,v).$

$U_{\text{snd}}(u) \leftarrow T(u).$

$Q_{\text{out}}(u,v) \leftarrow S(u,v), U(\_).$

Assume now a restricted set of keys $\mathcal{K}$. We have that, whichever (restricted) $\mathcal{K}$ we chose, the related specification might no longer be convergent. To see why this is the case, consider an initial instance $\mathbf{I}$ and $\mathcal{K}$ maximal, i.e., every term of every relation is key[14]. Assume $\mathbf{I}$ such that $\text{adom}(I_R) \supset \text{adom}(I_T)$, and a configuration in which N is large. In this situation, it may well happen that a non-empty set of facts in I_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 $\mathcal{Q}(\mathbf{I})$. Thus, this transducer is not convergent.

4.4.1 Connected queries

A class of monotonic queries exists which is restricted: *connected queries* (Guessarian 1990). Informally, a query is connected if every relation in a rule-body is connected through a join-path with every other relation composing the same rule-body.

[14] We chose $\mathcal{K}$ to be maximal because if we fail to generate a convergent specification for the maximal case, even more so specifications where $\mathcal{K}$ is less than maximal will fail.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Definition 9
Let $body(q_R)$ be the conjunction of literals defining the body of a rule q_R. We say that two different literals $R_i(\bar{u}_i)$, $R_j(\bar{u}_j) \in body(q_R)$ are *connected* in q_R if either

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

Two relations R_i and R_j are said to be connected in q_R if there are two literals $R_i(\bar{u}_i)$ and $R_j(\bar{u}_j)$ that are connected in q_R.

Definition 10
We say that a $\mathscr{L}$-query $\mathscr{Q}$ is *connected* if there is no rule $q_R \in \mathscr{Q}$ whose body contains either a nullary relation or a relation R_i which is not connected with any other relation $R_j \in body(q_R)$.

Remarks: (i) Literals can be either positive or negative predicates[15]; (ii) every positive query composed by a single rule containing just one non-nullary body atom is connected by definition; and (iii) every non-nullary unconnected query is an *existential query* (Ramakrishnan *et al.* 1988).

Example 9
The following DATALOG query returns all the nodes being source of a triangle, if a non-cyclic path of length four exists. In this query, the build-it predicate $\neq$ is used (in infix notation) to express inequality between variable instantiations.

$$T(u) \leftarrow E(u,v), E(v,w), E(w,u).$$
$$F() \leftarrow E(u,v), E(v,w), E(w,x), E(x,y), x \neq u, y \neq u.$$
$$Q(u) \leftarrow T(u), F().$$

While the first two rules are connected, the third rule is not. The query is therefore not connected.

Proposition 3
Let $\mathscr{L} \subseteq$ DATALOG be a language. For every connected (monotonic) query expressible in $\mathscr{L}$, an equivalent one exists that is restricted.

Proof
From Proposition 2, we know that a specification $\mathscr{N}$ exists parallelly computing every monotonic query $\mathscr{Q}$ for all input instances $\mathbf{I}$. Starting from $\mathscr{N}$, we can construct a new specification $\mathscr{N}'$ where every rule in Q_{out} is primed and moved to Q_{snd}. We then add to Q_{out} a rule to output every fact over the output schema $\mathscr{D}_{\text{out}}$. The behavior of $\mathscr{N}'$ 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 $\mathscr{N}'$ because $\mathscr{K}$ is unrestricted and, for this reason, every query is live. Every fact in $\mathscr{Q}(\mathbf{I})$, and no more, must hence also be in $\mathscr{N}'(\mathbf{I})$ whichever configuration we chose since the query is monotonic.

[15] Note that non-nullary negative literals are connected by definition since we are only considering safe queries.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Consider $\mathscr{CN}'$, the class of correct specifications defined by $\mathscr{N}'$. Assume now a new specification $\mathscr{N}''$ derived from $\mathscr{N}'$ by considering a restricted keys-set $\mathscr{K}''$. For simplicity, we fix $\mathscr{K}''$ to be maximal. We have to show that $\mathscr{N}''$ is eventually consistent with $\mathscr{N}'$. This is quite straightforward: every query is connected thus the liveness property still holds because $\mathscr{K}''$ is maximal and hence a non-null intersection exists among the destinations of all the atoms composing every rule-body. Reasoning in the same way, every rule is also evaluated on a instance. For what concerns safety, this is trivially satisfied because the query is monotonic.

We are now going to show how an inflationary transducer $\mathscr{T} = (\mathscr{P}, \Upsilon, \mathscr{K}'')$ composing the specification $\mathscr{N}''$ can be built. Let $\Upsilon_{\text{db}} = \mathscr{D}_{\text{in}}$, $\Upsilon_{\text{snd}} = \{R' \mid R \in \text{sch}(\mathscr{Q})\}$, $\Upsilon_{\text{out}} = \mathscr{D}_{\text{out}}$ (*systems* and *time* relations are as usual), and every term in every *communication* relation is key. Since $\mathscr{Q}$ is monotonic, no result can be derived that will be retracted in the future. The idea was to apply every rule as it is, and every new derived fact is sent to the other nodes composing the network. Concretely, we first add to Q_{snd}, for each $R \in \Upsilon_{\text{db}}$, the following rule implementing the shuffling:

$$R'_{\text{snd}}(\bar{u}) \leftarrow R(\bar{u}). \tag{6}$$

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

$$R_{\text{out}}(\bar{u}) \leftarrow R'(\bar{u}) \tag{7}$$

for each relation $R \in \mathscr{D}_{\text{out}}$ to output the results. Note that the transducer is oblivious and monotonic. Is easy to see that a transducer program generated in this way computes the initial query $\mathscr{Q}$. □

Connected queries have an interesting semantic property: they *distribute over components* (Ameloot *et al.* 2014). An instance $\mathbf{J}$ is said to be *connected* if whichever pair of constants $\mathtt{a}, \mathtt{b} \in \mathit{adom}(\mathbf{J})$, a chain of facts $\mathbf{f}_1, \ldots, \mathbf{f}_n$ exists such that $\mathtt{a} \in \mathit{adom}(\mathbf{f}_1)$, $\mathtt{b} \in \mathit{adom}(\mathbf{f}_n)$, and for any pair of consecutive fact $\mathbf{f}_i$, $\mathbf{f}_{i+1}$ in the chain with $0 \leqslant i < n$, $\mathit{adom}(\mathbf{f}_i) \cap \mathit{adom}(\mathbf{f}_{i+1}) \neq \emptyset$. Now, if $\mathbf{I}$ is an instance, $\mathbf{J}$ is a *component* of $\mathbf{I}$ if (i) $\mathbf{J} \subseteq \mathbf{I}$, (ii) $\mathbf{J}$ is non-empty, and (iii) $\mathbf{J}$ is connected and maximal with this property in $\mathbf{I}$. Finally, if with $co(\mathbf{I})$ we denote the components of $\mathbf{I}$, a query $\mathscr{Q}$ is said to *distribute over components* if $\mathscr{Q}(\mathbf{I}) = \bigcup_{\mathbf{J} \in co(\mathbf{I})} \mathscr{Q}(\mathbf{J})$ for all $\mathbf{I}$.

Proposition 4 (*Ameloot at al. 2014*)
Let $\mathscr{L} \subseteq$ DATALOG$^{\neg}$. Every connected $\mathscr{L}$-query distributes over components.

Proposition 5 (*Ameloot et al. 2015*)
Every query computable by a DATALOG$^{\neg}$-query that distributes over components can be computed by a connected DATALOG$^{\neg}$-query.

Example 10
Consider again Example 9. The query is not connected and in fact it does not distribute over components. To see why this is the case, assume that the input instance is composed by two components. The query does not distribute because the result on each component might depend on the presence of a path of length four on the other component.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Conversely, assume the query:

$$T(u) \leftarrow E(u,v), E(v,w), E(w,u)$$
$$F(u) \leftarrow E(u,v), E(v,w), E(w,x), E(x,y), x \neq u, y \neq u$$
$$Q() \leftarrow T(u), F(u)$$

returning true if a component exist having a triangle, with a path of length four starting from the same source node. The query is connected and distributes over components.

Remark: Differently from Ameloot *et al.* (2015), but according to Ameloot *et al.* (2014), we are considering as not connected *all* the queries having nullary relations in the body. In Ameloot *et al.* (2015), a specific type of nullary relations is identified which can be safely used in rule bodies while maintaining connectedness. Such nullary relations are however copied to each connected component, which in our parallel settings means broadcasting all nullary relations, which in turn means that, for partitioned hash families, if negation is applied over a nullary relation the query is no longer parallelly computable (because unsafe). We therefore restrict our attention to non-empty instances (having at least one component) and connected queries without nullary atoms in the body.

4.4.2 Non-monotonic queries

Clearly, unconnected non-monotonic queries are not restricted. Interestingly, not every connected non-monotonic query is however restricted.

Example 11
Consider the following DATALOG$^{\neg}$-query computing the facts in T not in the transitive closure of R.

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

This query can be parallelly computed by the following oblivious and inflationary FO-transducer:

Schema: $\Upsilon_{\text{db}} = \{R^{(2)}, T^{(2)}\}$, $\Upsilon_{\text{com}} = \{S^{(1,2)}, U^{(1,2)}, CS^{(1,2)}\}$,
$\Upsilon_{\text{mem}} = \{\texttt{Ready}^{(0)}\}$, $\Upsilon_{\text{out}} = \{Q^{(2)}\}$

Program:
$$S_{\text{snd}}(v,u) \leftarrow R(u,v).$$
$$U_{\text{snd}}(u,v) \leftarrow T(u,v).$$
$$CS_{\text{snd}}(u,v) \leftarrow S(u,v).$$
$$CS_{\text{snd}}(u,w) \leftarrow S(u,v), CS(v,w).$$
$$\texttt{Ready}_{\text{ins}}() \leftarrow \neg\texttt{Ready}().$$
$$Q_{\text{out}}(u,v) \leftarrow U(u,v), \neg CS(u,v), \texttt{Ready}().$$

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

This specification correctly computes the query only when some specific hashing functions are used. Indeed, it might happen that facts are distributed unevenly among the nodes, and that a node ends up deriving a new fact over CS after all the other nodes have already finished their computation. This may result in the possibility that a fact over Q be retracted: Intuitively, the problem is that negation is applied too early. In order to avoid this situation, common knowledge of local termination for each node is required, i.e., nodes must *synchronize*: Every node should notify every other node that it has locally terminated the computation of CS and then, when every node has locally terminated the computation of the closure of R, and all nodes know this, Q can be safely evaluated. Clearly, this pattern requires a non-restricted 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. This again is obtainable only with an unrestricted specification.

Nevertheless, a class of non-monotonic queries exists that does not require broadcasting rules: *recursion-delimited connected* queries.

Definition 11
Given a DATALOG$^\neg$ query $\mathcal{Q}$, we say that $\mathcal{Q}$ is *recursion-delimited* if, whichever stratification $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ we choose, $\mathcal{Q}_1, \ldots, \mathcal{Q}_{n-1}$ are non-recursive programs, while $\mathcal{Q}_n$ is expressed in $\mathcal{L}$, with $\mathcal{L} \subseteq$ DATALOG$^\neg$, and negation is applied over extensional database schemas only[16].

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

Proposition 6
Let $\mathcal{L} \subseteq$ DATALOG$^\neg$. Every recursion-delimited, connected query expressible in $\mathcal{L}$ is restricted.

Proof
We follow the same procedure presented in the proof of Proposition 3. Let $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ be a stratification of the rules of the input query $\mathcal{Q}$. 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 $\mathcal{D}_{\text{out}}$ is then output. Since all the rules are connected, and the set of keys is maximal, the liveness property is always satisfied so we have the same opportunity of deriving fact as in the case in which $\mathcal{K}$ is unrestricted. Although the safety property is not satisfied, no wrong result can be inferred because each stratum is evaluated sequentially and every rule is connected end evaluated on a instance. We can then conclude that the specification $(\mathcal{T}, \mathcal{T}^e, \gamma)$ parallelly computes $\mathcal{Q}$, and $\mathcal{Q}$ is a restricted query. More precisely, initially set $\Upsilon_{\text{db}} = \mathcal{D}_{\text{in}}$, $\Upsilon_{\text{snd}} = \{R' \mid R \in \text{sch}(\mathcal{Q})$, $\Upsilon_{\text{out}} = \mathcal{D}_{\text{out}}$ (*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_{snd} a rule in the form of equation (6) for each $R \in \Upsilon_{\text{db}}$ to

[16] This language is commonly referred to as *semi-positive* DATALOG$^\neg$ (see also Appendix B.1).

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

implement the shuffling (similarly to the proof of Proposition 3). Since $\mathcal{Q}$ is non-monotonic, an appropriate order of evaluation of rules must be enforced if we do not want to derive wrong results. Thus, consider the stratification $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ of $\mathcal{Q}$. First, consider the first $n-1$ strata. By definition of recursion-delimited 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 that depends on another predicate belongs to a higher stage. Intuitively, the stratification is maintained since all predicates belonging to a higher stratum also belong to a higher stage. We then have that stages, basically, are used to implement distributed stratification. Let m the highest stage thus obtained. We create a new stage $m+1$ containing all the predicates in the last stratum $\mathcal{Q}_n$. Consider now the query $\mathcal{Q}'$ obtained from $\mathcal{Q}$ by (i) priming all relations and (ii) appending to the body of each rule in $\mathcal{Q}_j$ a nullary atom $\texttt{Stage}^j()$, with $j \in 1, \ldots, m+1$, in order to bind the evaluation of rules to the respective stage. We now add to Q_{snd} all the queries in $\mathcal{Q}'$, and to Q_{out} a rule in the form of equation (7) for each *output* relation. Finally, in order to advance stage by stage, we add one rule in the form:

$$\texttt{Stage}^j_{\text{ins}}() \leftarrow \texttt{Stage}^i() \tag{8}$$

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

$$\texttt{Stage}^1_{\text{ins}}() \leftarrow \neg\texttt{Stage}^1(), \neg\texttt{Stage}^2(), \ldots, \neg\texttt{Stage}^n \tag{9}$$

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 $\mathcal{Q}$. The main difference with respect to the proof of Proposition 3 is that now the query is non-monotonic and therefore each negative literal cannot be evaluated before all the related tuples are generated 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 rules having just extensional relations of $\mathcal{Q}$ in the body are evaluated. Let us denote with q_R one of such rules. Since every previously sent fact is hashed over all the terms composing the tuple, and since every rule is connected, we have that at least a node which is able to satisfy $body(q_R)$ exists, and a set of facts will be sent. No wrong tuple can be derived because every rule is connected. All the newly derived intensional facts, plus the previous extensional tuples, are now sent to a set of nodes based on the parallelization strategy. The queries of the successive stage will then be evaluated in the successive round, and again, by construction, they are all evaluated on a instance. Let now assume we are at stage m and that ever query has been evaluated sequentially on a instance until that point. Again this means that a new set of facts will be sent, together with the previously sent 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 is allowed to be recursive. We finally have that every rule in Q is evaluated on an instance by construction. This concludes the proof. □

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Example 12
Let $\mathcal{Q}$ be the following recursion-delimited, connected query:

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

which computes a transitive closure applied over a filtered set of edges. A FO-transducer parallelly computing the query is the following[17]:

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

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

Remarks: (i) Monotonic queries do not have the same problem as non-recursion-delimited queries: Even if tuples are hashed unevenly, no retraction can occur because of monotonicity. (ii) If a mechanism existed for which a recursive DATALOG query could be rewritten in a non-recursive form, all connected 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, is undecidable (Gaifman *et al.* 1993).

Corollary 1
Every restricted query is connected.

Proof
The fact that every restricted query computes a connected one directly follows from the definition of connectedness. Conversely a restricted query cannot be unconnected as shown for instance in Examples 8 and 11. □

Figure 2 depicts the query taxonomy as discussed so far.

5 Coordination-freedom refined

We have seen in Section 3.6 that, for synchronous and reliable systems, a particular notion of coordination-freedom is needed. In fact, we have shown that certain non-monotonic queries – e.g., Example 5 – that under the asynchronous model

[17] Note that nullary relations $\texttt{Stage}^i$ used to postpone the evaluation of rules can be actually omitted for this specific example: For clarity, we however follow the technique used in the proof of Proposition 6.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

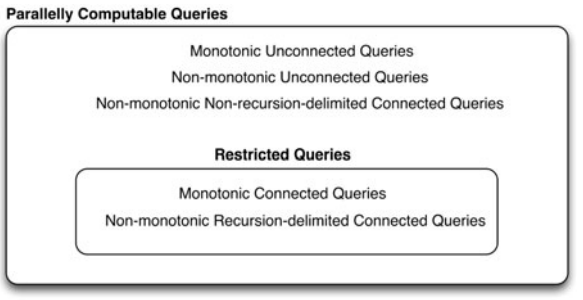

Fig. 2. Query taxonomy.

require coordination can be computed in a coordination-free way. The key-point is that, as observed in Ameloot *et al.* (2013), in asynchronous systems coordination-freedom is directly related to communication-freedom under ideal distribution. That is, if the distribution is right, no communication is required to correctly compute a coordination-free query because (i) no data must be sent (the distribution is correct) and (ii) no "control message" is required to obtain the correct result (the query is coordination-free). However, due to its synchronous nature, in *bsp* settings non-monotonic queries can be computed in general without resorting to coordination. As already anticipated, this is due to the fact that the above concept of coordination is already "baked" into the *bsp* model: Each node is synchronized with every other one, hence "control messages" are somehow implicitly assumed. In this section, we will introduce a novel knowledge-oriented perspective linking coordination with the way in which explicit and implicit information flow in the network (Contribution 3). Under this new perspective, we will see that a specification needs coordination if, in order to maintain convergence, a node must have *some form of information exchange* with all the other nodes.

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 the existence of primitives enforcing some control over the ordering of events (Hunt *et al.* 2010). In a seminal paper (Lamport 1978), Lamport proposed a synchronization algorithm exploiting the 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'[18], and thus 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) has been recently proposed (Ben-Zvi and

[18] The potential causality relation is often quoted as *happened-before*.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Moses 2014) to generalize Lamport's potential causality to synchronous systems. Using syncausality, we are able to model how information flows among nodes with the passing of time. Given a run ρ and two points (cf. Section 3.2) (ρ^i, t), (ρ^j, t') for not necessarily distinct nodes $i, j \in N$, we say that (ρ^j, t') causally depends on (ρ^i, t) if either $i = j$ and $t \leqslant t'$ – i.e., a local state depends on the previous one – or a tuple has been emitted by node i at time t, and received by node j, with $t < t'$[19]. We refer to these two types of dependencies as *direct*.

Definition 12
Given a generic system $\mathscr{S}$, and a run $\rho \in \mathscr{S}$, we say that two points (ρ^i, t), (ρ^j, t') are related by a *direct potential causality* relation $\rightarrow$, if at least one of the following is true:

(1) $t' = t + 1$ and $i = j$.
(2) $t' \geqslant t + 1$ and node j receive a tuple at time $t + 1$ which was sent by i at time t.
(3) There is a point (ρ^k, t'') s.t. $(\rho^i, t) \rightarrow (\rho^k, t'')$ and $(\rho^k, t'') \rightarrow (\rho^j, t')$.

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

Differently from asynchronous systems, we however have that a point in node j can occasionally *indirectly* depend on another point in node i even if no fact addressed to j is actually sent by i. This is because j can still draw some conclusion simply as a consequence of the *bounded delay guarantee* and *deterministic delivery* (**S3**) of synchronous systems. That is, each node can use the common knowledge that every sent tuple is received at most after a certain bounded delay to reason about the state of the system. The bounded delay guarantee can be modeled as an imaginary `NULL` fact, as in Lamport (1984). 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 negation can be safely applied to a predicate. In general, negation can be applied to a literal $R(\bar{u})$ when the content of R is *sealed* for what concerns the current round. In local settings, we have that such condition holds for a predicate at round t' if its content has been completely generated at round t, with $t' > t$. In distributed settings, we have that, if R is a *communication* relation, being in a new round t' 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 assured that every tuple has been delivered, the above condition does not hold. The result is that negation cannot be applied safely[20].

We will model the fact that, in synchronous systems, the content of a *communication* relation R is stable because of the bounded delay guarantee **S3**, by having every node i emit a fact $\texttt{NULL}^i_R$ at round t, for every *communication* relation R, then each

[19] Note that a point in a synchronous system is what Lamport defines as an event in an asynchronous system.
[20] Note that we can reason in the same way also for every other negative literal depending on R. For this reason, here we consider only the case in which negation is only applied over *communication* relations.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

$\texttt{NULL}^i_R$ fact will be delivered to node j exactly by the next round. We thus have that the content of R is stable once j has received a $\texttt{NULL}^i_R$ fact from every $i \in N$. The sealing of a *communication* relation at a certain round is then ascertained only when $|N|$ $\texttt{NULL}_R$ facts have been counted. We call this *Snapshot Closed World Assumption* (*SCWA*): Negation on relation R can be applied by a node just when it has surely received a consistent snapshot (Babaoğlu and Marzullo 1993) of the global instance I_R.

We can assume that the program generating NULL facts is adjoined to the transducer program. Recall however that not necessarily the $\texttt{NULL}^i_R$ facts must be physically sent: This in particular is true under the deterministic delivery semantics of *bsp*, where the strike of a new round automatically seals all the *communication* relations. In other words, under *bsp* the program generating NULL facts is *virtual*: No rule is fired and no fact is actually sent because the system definition already implicitly assume SCWA. Still these virtual facts will help us in reasoning about indirect flows of information, therefore we will still assume that such NULL facts are "virtually" derived and sent. Example 13 below shows a concrete situation showing that although no NULL fact is sent, still SCWA holds. The reader however should not believe that this is always the case. In fact, in Section 6.2, we will see how by simply dropping the deterministic delivery property **S3′** the situation becomes more complicated, and the virtual program generating NULL facts must actually be evaluated. As a final remark, note that no $\texttt{NULL}^i_R$ fact need be issued if no SCWA must be enforced over R or over a dependent relation.

Definition 13
Given a run ρ, we say that two points (ρ^i, t), (ρ^j, t') are related by an *indirect potential causality* relation , if $i \neq j$, $t' \geqslant t+1$ and a $\texttt{NULL}^i_R$ fact addressed to node j has been (virtually) sent by node i at round t.

Example 13
Consider the program of Example 5, a proper configuration and an initial instance. At round $t+1$, 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 sent. Despite this, every node can still conclude at round $t+1$ that the content of S is stable. In this situation, we clearly have an indirect potential causality relation.

Corollary 2
A necessary condition for an indirect potential causality relation to exist is the presence of a negated literal.

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 14
Let ρ be a run in the synchronous system $\mathscr{S}^{\text{rsync}}$. The *syncausality* relation $\rightsquigarrow$ is the smallest relation such that

1. if $(\rho^i, t) \rightarrow (\rho^j, t')$, then $(\rho^i, t) \rightsquigarrow (\rho^j, t')$;

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

2. if $(\rho^i, t) \;\; (\rho^j, t')$, then $(\rho^i, t) \rightsquigarrow (\rho^j, t')$; and
3. if $(\rho^i, t) \rightsquigarrow (\rho^j, t')$ and $(\rho^j, t') \rightsquigarrow (\rho^k, t'')$, then $(\rho^i, t) \rightsquigarrow (\rho^k, t'')$.

5.2 From syncausality to coordination

We next propose the *predicate-level syncausality* relationship, modeling causal relations at the predicate level. That is, instead of considering how (direct and indirect) information flows between nodes, we introduce a more fine-grained relationship modeling the flows between *predicates and nodes.*

Definition 15
Given a run $\rho \in \mathscr{S}^{\text{rsync}}$, we say that two points (ρ^i, t), (ρ^j, t') are linked by a relation of *predicate-level syncausality* $\overset{R}{\rightsquigarrow}$, if any of the following holds:

1. $i = j$, $t' = t + 1$ and a tuple over $R \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$ has been derived by a query in $Q_{\text{ins}} \cup Q_{\text{out}}$.
2. $R \in \Upsilon_{\text{com}}$ and node i sends a tuple over R at time t addressed to node j, with $t' \geqslant t + 1$.
3. $R \in \Upsilon_{\text{com}}$ and node i (virtually) sends a $\texttt{NULL}^i_R$ fact at time t addressed to node j, with $t' \geqslant t + 1$.
4. There is a point (ρ^k, t'') s.t. $(\rho^i, t) \overset{R}{\rightsquigarrow} (\rho^k, t'')$ and $(\rho^k, t'') \overset{R}{\rightsquigarrow} (\rho^j, t')$.

We are now able to specify a condition for achieving coordination. Informally, we have that *coordination exists when all the nodes of a network reach a common agreement that some event has happened*. But the only way to reach such agreement is that an (direct or indirect) information flow exists between the node in which the event actually occurred, and every other node. This is a sufficient and necessary condition because of the reliability and bounded delay guarantee of *rsync* system. We formalize this intuition using the (predicate level) syncausality relationship:

Definition 16
We say that a correct specification class $\mathscr{CN}$ *manifests the coordination pattern* if, for all possible initial instances $\mathbf{I} \in \text{inst}(\Upsilon_{\text{db}})$, whichever run $\rho \in \mathscr{S}_{\mathscr{CN}}(\mathbf{I})$ we select where N is not trivial, a point (ρ^i, t) and a *communication* relation R exist so that $\forall j \in N$ there is a predicate-level syncausality relation with $(\rho^i, t) \overset{R}{\rightsquigarrow} (\rho^j, t')$ and $t' \leqslant *$.

We call node i the *coordination master*. A pattern with a similar role has been named *broom* in Ben-Zvi and Moses (2011). Note that the condition $t' \leqslant *$ is needed since only those points which actually contribute to the definition of the final state are of interest, while all the others can be ignored.

Remark: The reader can now appreciate to which extent coordination was already "baked" inside the broadcasting synchronous specifications of Section 3. Note that broadcasting, in *bsp*, brings coordination. This is not necessarily true in asynchronous systems.

Intuitively, the coordination master is where the event occurs. If a broadcasting of (informative or non-informative) facts occurs, then such event will become common

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

knowledge among the nodes. On the contrary, if broadcasting is not occurring, common knowledge cannot be obtained, therefore if the correct final outcome is still reached, this is obtained without coordination. That is, if at least a non-trivial configuration exists s.t. the coordination pattern does not manifest itself, we have coordination-freedom:

Definition 17
Given a correct class $\mathscr{CN}$ and an initial instance $\mathbf{I}$, we say that $\mathscr{CN}$ is *coordination-free* if a non-trivial configuration (N, D, H) can be selected for which $\mathscr{S}_{\mathscr{N}}(N, D, H, \mathbf{I})$ does not manifest the coordination pattern, where $\mathscr{N} \in \mathscr{CN}$.

W.l.o.g., we will also dub the specifications belonging to $\mathscr{CN}$ as *coordination-free*. From Definition 17, we can deduce the following proposition:

Proposition 7
Every coordination-free specification parallelly computes a restricted 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 *communication* relation is allowed to have a key set to 0 because no broadcasting must occur. □

The reverse result clearly does not hold: A restricted query might require coordination, e.g., non-monotonic, connected, recursion-delimited queries are restricted and not coordination-free.

Note that our definition of coordination-freedom based on syncausality relation makes it rather intuitive, in contrast with the original, declarative definition of Ameloot *et al.* (2013).

5.3 From coordination-freedom to communication-freedom

Ameloot *et al.* (2013) relates coordination-freedom with absence of communication under ideal distribution. It would then also be interesting to explore which relationship exists between our definition of coordination-freedom and communication. Since in a coordination-free specification, broadcasting queries are not strictly needed, we can deduce that every coordination-free specification can be made communication-free. That is, at least a configuration exists for which the correct result is computed without emitting any fact: The trivial case is the configuration in which the partition function installs the full initial instance on one node only, and H addresses every constant to the same node.

Example 14
As an example of a coordination- and communication-free specification, consider the following UCQ-network $\mathscr{N}$ computing the transitive closure of the relation $R^{(2)}$: Each node computes the closure of R on its local data and then emits the derived atoms.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

$$\begin{aligned}\text{Schema: } & \Upsilon_{\text{db}} = \{R^{(2)}\}, \Upsilon_{\text{com}} = \{S^{(1,2)}\}, \Upsilon_{\text{out}} = \{T^{(2)}\}\\ \text{Program: } & T_{\text{out}}(u,v) \leftarrow R(u,v).\\ & T_{\text{out}}(v,w) \leftarrow S(u,v), T(u,w).\\ & S_{\text{snd}}(u,v) \leftarrow T(u,v).\end{aligned}$$

$\mathcal{N}$ is oblivious and $\mathcal{P}^{\text{bsp}}_{\mathcal{N}}(\mathbf{I})$ is convergent for every initial instance $\mathbf{I}$. Since there is no negation, we have only to show that for every initial instance, a configuration exists such that the sending queries are not broadcasting. Consider D and H such that the full instance is installed on one node, and every constant is hashed to that the same node. $\mathcal{N}_{N,D,H}$ is communication-free.

We can therefore deduce that coordination-freedom might be a sufficient condition for a specification to be communication-free; however, it is not a necessary condition, as shown by the next example.

Example 15
Let $\mathcal{Q}$ be the following non-monotonic query:

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

The following FO-transducer network parallelly computes $\mathcal{Q}$:

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

The specification is non-monotonic (and thus requires coordination), restricted, and can be made communication-free. Consider in fact a distribution function D 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, and non-trivial N. Whichever initial instance $\mathbf{I}$ is given, we clearly have that $\mathcal{N}_{N,D,H}$ is communication-free.

Unfortunately, coordination-freedom is undecidable in general[21]. However, from the above intuitions, we can draw the following:

Proposition 8
Every restricted query is parallelly computable by a hashing specification which can be made communication-free.

[21] Recall that for a specification to be coordination-free, it must first of all be independent. Independence is undecidable (cf. Appendix A).

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Proof
By definition, every restricted query is parallelly computable by a specification $\mathscr{N}$. We have to show that $\mathscr{N}$ can be made communication-free. Consider a configuration where N is non-trivial and D, H are such that the full initial instance is installed on node i, and a hash function exists so that every constant is hashed to the same node i. We have that $\mathscr{N}_{N,D,H}$ is independent by definition, and communication-free. □

Corollary 3
Every coordination-free specification is communication-free.

With Proposition 8, we have described one of the characteristics of restricted queries: they are communication-free. In the next section, we will see that a larger class of queries can be computed in a communication-free way. These will be called *embarrassingly parallel queries*[22].

6 CALM in Rsync systems

As we have seen, the original version of the CALM principle as postulated in Conjecture 1 is trivially not satisfiable in *bsp* systems, because some monotonic queries exist – i.e., unconnected queries, cf. Section 4.3 and Example 8 – that are not coordination-free in the sense of Definition 17. We will then prove the CALM conjecture just for the remaining class of monotonic queries. In the remainder of the section, we will then close the circle by discussing how the notion of coordination introduced in Ameloot *et al.* (2013) collapses into the one we proposed, once the synchronization constraints are weakened.

6.1 *The CALM conjecture for* bsp *systems*

Let us first introduce the following lemma:

Lemma 2
Let $\mathscr{L} \subseteq$ DATALOG$^{\neg}$ be a query language. Every query that is parallelly computed by a coordination-free $\mathscr{L}$-transducer network is monotone and connected.

Proof
Let $\mathscr{N}$ be a coordination-free $\mathscr{L}$-transducer network parallelly computing a query $\mathscr{Q}$. By applying Proposition 7 and Corollary 1, we know that $\mathscr{Q}$ is connected, but might still be non-monotonic (cf. Figure 2). It remains then to show that $\mathscr{Q}$ is indeed monotonic. Let $\mathbf{I}$ and $\mathbf{J}$ be two initial instances over $\mathscr{D}_{\text{in}}$ such that $\mathbf{I} \subseteq \mathbf{J}$. We have to show that $\mathscr{Q}(\mathbf{I}) \subseteq \mathscr{Q}(\mathbf{J})$. Assume first that $\text{adom}(\mathbf{J} \setminus \mathbf{I}) \cap \text{adom}(\mathbf{I}) = \emptyset$. Our intuition is that, under this assumption, a configuration exists such that communication is not required to prove monotonicity. We will then use such configuration to show that indeed this holds also for any arbitrary pair of instances. Since $\mathscr{N}$ parallelly computes $\mathscr{Q}$, $\mathscr{N}$ is independent, therefore a non-trivial configuration exists such that

[22] 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 (Foster 1995).

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

D installs $\mathbf{I}$ on one single node, while $\mathbf{J} \setminus \mathbf{I}$ is completely installed in another node. In addition, assume a hash function in H such that all the constant in $\mathbf{I}$ are hashed to the same node i in which $\mathbf{I}$ is installed, while all the constant in adom$(\mathbf{J} \setminus \mathbf{I})$ are hashed to the node $j \neq i$ in which $\mathbf{J} \setminus \mathbf{I}$ resides. Consider a fact $\mathbf{f} \in \mathscr{Q}(\mathbf{I})$. By construction, $\mathbf{f}$ will appear in the output of $\mathscr{N}_{N,D,H}(\mathbf{I})$ at node i at a certain round t. Consider now the case in which the input instance is $\mathbf{J}$. We have a node $j \neq i$ such that $I^j_{\text{db}} = \mathbf{J} \setminus \mathbf{I}$, while again $I^i_{\text{db}} = \mathbf{I}$. Let us consider the point (ρ, t) of the run $\rho \in \mathscr{S}^{\text{bsp}}_{\mathscr{N}}(N, D, H, \mathbf{I})$. 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 also in run ρ. Again, by construction, being $\mathscr{N}$ independent, $\mathscr{N}_{N,D,H}(\mathbf{J})$ parallelly computes the query $\mathscr{Q}(\mathbf{J})$, therefore $\mathbf{f}$ must also belong to $\mathscr{Q}(\mathbf{J})$.

Consider now the communication-free specification $\mathscr{F}$ built from $\mathscr{N}$: We can freely use this procedure since $\mathscr{N}$ is communication-free by Proposition 8. We then have that $\mathscr{N}_{N,D,H}(\mathbf{I}) = \mathscr{F}_{N,D,H}(\mathbf{I})$ and, similarly, $\mathscr{N}_{N,D,H}(\mathbf{J}) = \mathscr{F}_{N,D,H}(\mathbf{J})$. Consider now two generic input instances $\mathbf{J}', \mathbf{I}'$ with $\mathbf{J}' \supseteq \mathbf{I}'$, and the same distribution function D installing $\mathbf{I}'$ on node i and $\mathbf{J}' \setminus \mathbf{I}'$ on node j. Also, in this case, we have that $\mathscr{F}_{N,D,H}$ parallelly computes $\mathscr{Q}$ and, reasoning as above, $\mathscr{F}_{N,D,H}(\mathbf{I}') \subseteq \mathscr{F}_{N,D,H}(\mathbf{J}')$, thus $\mathscr{F}$ is monotonic. As a consequence, $\mathscr{N}$ is also monotonic. □

We are now able to prove the restricted version of the CALM conjecture for *bsp* systems (Contribution 4):

Theorem 1
A query can be parallelly computed by a coordination-free transducer network iff it is monotone and connected.

Proof
Starting from the if direction, by Proposition 3, we know that a connected DATALOG (i.e., monotonic) query can be parallelly computed by an oblivious hashing transducer network $\mathscr{N} \in \mathscr{CN}$. It remains to show that $\mathscr{N}$ is coordination-free. We can notice that, because the transducer network is monotonic, no coordination pattern can occur because of indirect information flow (from Corollary 2). On the other hand, a coordination pattern might occur because of direct information flow caused by broadcasting rules. Let $\mathscr{N}' \in \mathscr{CN}$ be a specification (not necessarily different from $\mathscr{N}$) such that $\mathscr{K}$ is restricted. Note that a restricted specification exists because every connected DATALOG query is restricted by Proposition 3. It remains to show that a non-trivial configuration (N, D, H) exists, such that for every initial instance $\mathbf{I}$, a run $\rho \in \mathscr{S}^{\text{bsp}}_{\mathscr{N}'}(N, D, H, \mathbf{I})$ exists where the coordination pattern does not 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 not broadcasted but addressed to the same node. Finally, $\mathscr{N} \in \mathscr{CN}$, therefore $\mathscr{N}_{N,D,H}$ correctly computes the query.

For what concerns the only-if direction, it is covered by Lemma 2.[23] □

[23] Note that the Lemma – and thus the Theorem – still holds although it is well known that a set of monotonic queries exists which are not expressible in DATALOG (Afrati *et al.* 1995).

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Corollary 4
A DATALOG query can be parallelly computed by a coordination-free transducer network iff it is monotone and distributes over components.

Proof
The corollary directly follows from Theorem 1 and Propositions 4–5. □

So far we have considered coordination-freedom. But what about communication-freedom? As previously mentioned, we name the class of communication-free queries as *embarrassingly parallel.*

Definition 18
Let $\mathscr{L}$ be a language and $\mathscr{Q}$ a $\mathscr{L}$-query. $\mathscr{Q}$ is *embarrassingly parallel* if it is parallelly computable by a specification that can be made communication-free.

As a preliminary answer to the above question, we can try to give a different reading of the CALM principle, by relating communication-freeness (instead of coordination-freeness) with monotonicity.

Lemma 3
Every oblivious specification parallelly 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 $\mathscr{N}$ computing a restricted monotonic query. We have already seen in Preposition 8 that a configuration exists which makes $\mathscr{N}$ communication-free. Consider now the case in which the monotonic specification $\mathscr{N}$ is computing a query $\mathscr{Q}$ which is not restricted. Consider a configuration (N, D, H) as described in Preposition 8: D installs the full instance on a unique node i, H addresses all the emitted facts to i, and N, is arbitrary but not trivial. $\mathscr{N}(N, D, H, \mathbf{I})$ is communication-free. We have to show that $\mathscr{N}(N, D, H, \mathbf{I})$ computes the query $\mathscr{Q}(\mathbf{I})$. Consider first all the nodes $j \neq i$ in N. Such nodes will output nothing since their instance is empty and the query is monotone. For what concern i, it exactly computes $\mathscr{Q}(\mathbf{I})$ since it contains the full instance. □

We can now state the following Theorem:

Theorem 2
Let $\mathscr{L}$ be a query language containing UCQ. For every query $\mathscr{Q}$ expressible in $\mathscr{L}$, the following are equivalent:

1. $\mathscr{Q}$ can be parallelly computed by an oblivious, inflationary transducer network.
2. $\mathscr{Q}$ is embarrassingly parallel.

Proof
$2 \Rightarrow 1$ follows from Proposition 3. It remains to prove that every oblivious and inflationary transducer can be made communication-free. We will show that the only kind of queries which can be parallelly computed and are not communication-free are the non-recursion-delimited queries. From Lemma 3, we already know that

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

monotonic queries are communication-free. From Proposition 8, we instead know that non-monotonic restricted recursion delimited queries also communication-free. We now proceed by contradiction: Assume $\mathcal{N}$ is a non-monotonic transducer network computing the query $\mathcal{Q}$, and $\mathcal{Q}$ is not recursion-delimited, 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, otherwise wrong facts could be derived. The transducer, in order to correctly compute the query, must therefore be able to detect when the recursion is terminated and hence the evaluation of the $m+1$th stratum can start. Since each node composing the network could end up having different (overlapping) partitions of the initial instance, different nodes might terminate the recursive computation in different rounds. Note that, although a partition might exist for which recursion terminates at the same round for all nodes, $\mathcal{N}$ is independent by definition, therefore it must be able to compute $\mathcal{Q}$ 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 node that it has finished its local computation. To express that a node has finished its computation, `id` must be clearly read, otherwise some receiving node might 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 know which nodes in the network have communicated that their local computation is completed. Clearly, this is not an oblivious specification since the *system* relations are employed. □

Discussion: Summarizing, we have seen that three different classes of coordination patterns can be identified under the *bsp* semantics (Contribution 5), all of them requiring acquisition of common knowledge of a property: *snapshot coordination*, which implements the SCWA, and require the common knowledge of a relation instance to be globally sealed; *broadcasting coordination* is required for unconnected queries and necessitate each node to know that a relation instance is not empty; and *synchronized coordination* requiring common knowledge of local termination of all the nodes. Broadcasting coordination is simple to implement because it only requires a broadcasting query. Snapshot coordination exploits the indirect information flow and hence is communication-free, and is used by any non-monotonic, recursion-delimited query. Finally, synchronized coordination necessarily requires access to *system* relations, since non-monotonic, non-recursion-delimited queries must be synchronized by a direct information flow in order to maintain consistency (cf. Example 11). Figure 3 updates Figure 2 with the new results we have just discussed.

An interesting difference among 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 deterministic delivery model (cf. Section 3.3), i.e., tuples

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

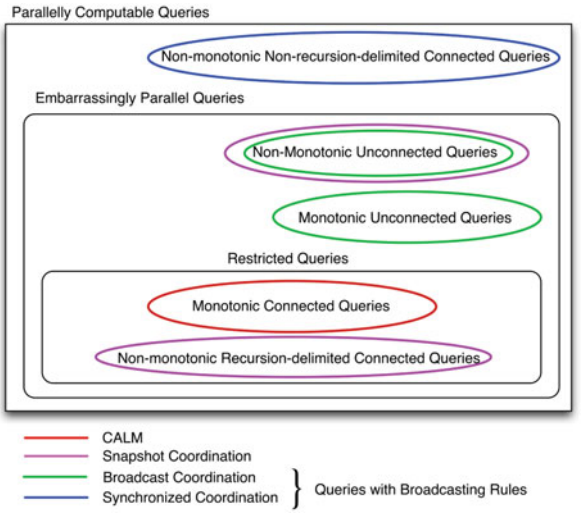

Fig. 3. Complete analysis of parallelly computable queries

arrive exactly after Δ physical time once emitted. But what happens if we assume less constrained systems, e.g., in MapReduce we start to pipeline Reducers with Mappers? Below, we first take a look at systems with non-deterministic delivery but bounded delay (Section 6.2, Mappers and Reducers tasks can be pipelined but in a single MapReduce step), and then conclude with *rsync* systems, i.e., systems with non-deterministic delivery and arbitrary, finite delay (Section 6.3, Mappers and Reducers are full pipelined).

6.2 Coordination and non-deterministic delivery

In Section 3.3, we have seen that *bsp* systems assume that all emitted messages arrive exactly after Δ physical time (i.e., condition **S3**′). In this section, we will instead assume that messages arrive non-deterministically within the Δ bound. Under this weaker condition, we are not any longer certain about when successive rounds can start: If we let rounds start after Δ physical time (as under *bsp* systems), we may spend unnecessary time waiting; conversely, if we start the next round right after all nodes have finished the current round (i.e., before Δ time has elapsed, hence before all messages are received with certainty), we may receive late facts and eventually have to retract wrong deductions.

Consider now an invertible function θ mapping each round number to the *physical time* in which it occurs, and two values Δ_{max}, Δ_{min} representing, respectively, the maximum and the minimum network latency of the given physical system. Let us simplify our model by substituting property **S3**′ with the following constraint, named *bounded delay*:

S3″ Let $del = \Delta_{max} - \Delta_{min}$, with $del \ll \theta(t+1) - \theta(t)$ for every pair of rounds $t, t+1$.

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

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

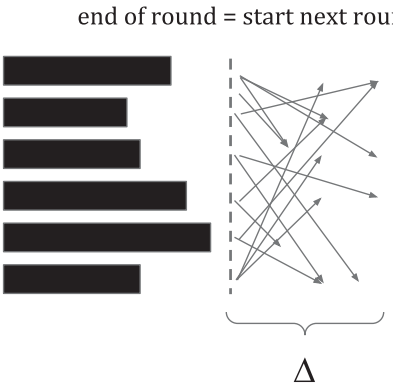

Fig. 4. bsp-d system computation model. Differently than *bsp* systems, the next round starts right after the end of the current one. Additionally, data communication may take arbitrary (although bounded) time.

assumptions, it follows that each tuple, derived by a send query at round t, will be available at the receiving site no later than the physical time $\theta(t+1)+\text{del}$, and that $\theta(t+1) \leqslant \theta(t+1)+\text{del} < \theta(t+2)$, i.e., facts are received during the successive round. Note that although the delay is bounded, and hence we are assured that every fact is delivered during the successive round, the actual instant in which a fact is received falls non-deterministically in the range $[\theta(t+1), \theta(t+1)+\text{del}]$. Henceforth, we will then use $\mathscr{S}^{\text{bsp-d}}$ to denote a *rsync* system with *bounded delay and non-deterministic delivery* (bsp-d). Figure 4 depicts how bsp-d systems behave, which is in line with frameworks such as MapReduce online (Condie *et al.* 2010), where key-value records are pipelined between Map and Reduce operations.

Given a synchronous transducer network $\mathscr{N}$, if we assume a synchronous system with non-deterministic delivery and bounded delay $\mathscr{S}^{\text{bsp-d}}$, 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 result 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 *bsp-d* and *bsp* systems. The same thing cannot be stated for non-monotonic programs, as the next example shows.

Example 16

Consider the transducer of Example 5 computing the emptiness query. Under non-deterministic delivery, it is not clear when negation can be safely applied over S. For instance, if negation is immediately applied, it may happen that a previously sent fact appears later, therefore invalidating the derived results.

In synchronous systems with non-deterministic delivery, we have that, in general, snapshot coordination is no more achievable "indirectly," without exchanging any message; under this model, we can therefore appreciate more in detail the nature of snapshot coordination.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

In order to explain how snapshot coordination can be implemented in *bsp-d* systems, we first solve the problem under the constraint that communication[24] implement *First In First Out* (*FIFO*) delivery[25], and then consider the general case. As a first step, we introduce how *monotonic and non-monotonic (stratified) aggregates* can be used in queries.

6.2.1 Queries with aggregates

Aggregate relations are usually employed in query languages to express *aggregate queries*. In the next subsections, we will use aggregate relations in positive rule-heads and in the form $R(\Lambda < \bar{w} >)$, with Λ one of the usual aggregate functions, and $\bar{w}$ a set of variables from the body (Ramakrishnan and Ullman 1995). Aggregate relations appear in the heads of *aggregation rules*:

$$R(\Lambda < \bar{w} >) \leftarrow B_1(\bar{u}_1), \ldots, B_n(\bar{u}_n). \tag{10}$$

If we denote with $\bar{\mathtt{W}}$ the finite multi-set containing all the existing ground assignments of $\bar{w}$ which satisfy the body of the rule, we have that $R(\mathtt{a})$ is true, where $\mathtt{a} = \Lambda < \bar{\mathtt{W}} >$. That is, $\mathtt{a}$ is the result of the application of Λ to the multi-set $\bar{\mathtt{W}}$ (Beeri *et al.* 1987).

We consider two different types of aggregate predicates: usual *stratified aggregates* and *monotonic aggregates* (Mazuran *et al.* 2013). For the former, they are stratified and hence the entire body must the stable (no holes are allowed in the local knowledge base) before the aggregate function can be applied. We then always assume the aggregate predicates to depend negatively on every predicate composing the body. This assumption is quite natural since head-aggregation rules can be easily rewritten as body-aggregation rules, which, in turn, can be specified using the stratified semantics of DATALOG$^{\neg}$ with built-in relations (Mumick and Shmueli 1995).

For what concerns the latter type of aggregates (the monotonic ones), since aggregation is monotonic, it is, for instance, allowed to appear in recursive rules. In order to differentiate between monotonic and stratified aggregation functions, we label the former with the *m* prefix. While we will not explain in detail the semantics behind monotonic aggregation – we suggest the interested reader to refer to Mazuran's paper or Zaniolo *et al.* (2016; 2017; 2018) for more recent evolvements – we nonetheless remark here the main operational differences between the two types of aggregation: stratified aggregation always returns a single value, which is the application of the function Λ over the stable multi-set $\bar{\mathtt{W}}$ once its computation is terminated; for monotonic aggregation, whenever the system is fed with new tuples, new values are returned forming a monotonically evolving distribution. For instance, if $m\_max < w >$ returns the max value of the term w, every time a new tuple is generated defining a new max value for w, $m\_max < w >$ will return it. Conversely, $max < w >$ will return the single maximum value for the stable multi-set $\mathtt{W}$.

[24] By communication here, we mean both the nodes' local buffers and the actual communication medium.
[25] Although non-deterministic delivery may look impractical with FIFO communication, this is a simplifying assumption by which facts are communicated in sequential order but where a bounded delay may occur between consecutive facts.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

6.2.2 Snapshot coordination under FIFO

Under the FIFO assumption, we have that tuples are received in the same order in which they are locally derived[26]. 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 (Babaoğlu and Marzullo 1993), and therefore apply one of the well-known snapshot protocols working under FIFO (Chandy and Lamport 1985).

Let $\mathscr{T}$ be a transducer used to parallelly compute a non-monotonic, recursion-delimited query $\mathscr{Q}$. In Section 5.2, we have seen that `NULL` messages are implicitly derived by send queries under the deterministic delivery assumption; by contrast, under non-deterministic delivery it might be necessary to explicitly send `NULL` messages. Consider a relation R occurring negated in the body of a rule in $\mathscr{Q}$. Let $\mathbf{B}$ be the body of the query $q_R \in Q_{\text{snd}}$ emitting R. We can add to Q_{snd} the following rules defining, respectively, a unary *stratified aggregate* relation $\texttt{CntR}^{(1)}$, and a new unary *communication* relation $\texttt{SealR}^{(0,1)}$ emulating the `NULL` message for R:

$$\texttt{CntR}(\text{count} < \bar{u} >) \leftarrow \mathbf{B}. \tag{11}$$

$$\texttt{SealR}_{\text{snd}}(i) \leftarrow \texttt{CntR}(u), \texttt{Id}(i). \tag{12}$$

In this way, by exploiting the stratified semantics, each node i can send the `NULL` message for relation R once the computation of the count of the number of tuples in R is completed. Since count is a stratified aggregate, `CntR` – and therefore also `SealR` – belongs to a higher stratum. In this way, we are assured that the `NULL` message is emitted after the instance over R has been completed. Under the FIFO semantics, we are then guaranteed that once a node receives a `SealR` tuple, the content of R is sealed for what concerns that emitting node. This clearly does not 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 can add to $\mathscr{T}$ the rules:

$$\texttt{CntSlR}(m\_count < u >) \leftarrow \texttt{SealR}(u), \tag{13}$$

$$\texttt{CntAll}(count < u >) \leftarrow \texttt{All}(u), \tag{14}$$

$$\texttt{FSR}() \leftarrow \texttt{CntSlR}(u), \texttt{CntAll}(u), \tag{15}$$

and we attach the final seal `FSR()` to the queries in which R is negated. Queries (13)–(15) are used to define when `FSR` is true, i.e., when n `NULL` messages have been received for relation R, with n the number of nodes in the network. Once `FSR` 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 `SealR`

[26] Note that this does not mean that tuples that are derived by different nodes in a certain global order are also received in the same order. FIFO can therefore be seen as enforcing a partial order.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

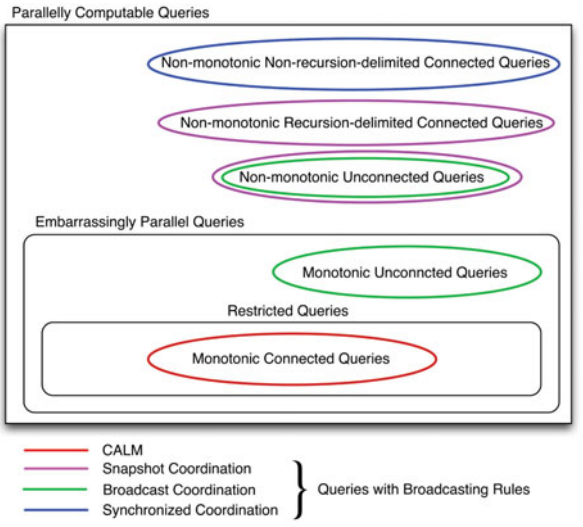

Fig. 5. Taxonomy for parallelly computable queries under *bsp-d*

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 *bsp* system to a *bsp-d* system, both *system* relations must be employed to implement snapshot coordination, and non-monotonic, connected, recursion-delimited queries are no longer embarrassingly parallel. This is consistent with Ameloot *et al.* (2013): non-monotonic queries are neither coordination- nor communication-free, and both `Id` and `All` relations are required. In *bsp-d* systems, we then have that syncausality degenerates into the Lamport's happens-before relation (Lamport 1978). Figure 5 depicts this new situation in which snapshot coordination code is injected into non-monotonic specifications.

6.2.3 Generic snapshot coordination

If we drop the FIFO assumption, we can end up in a situation in which a `NULL` message is received before a regular (informative) fact, therefore negation can end up being applied to a non-stable relation. Indeed this problem is related to how a partial order of events can be enforced in distributed settings (Lamport 1978). 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 `FSR` 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* (Babaoğlu and Marzullo 1993) of ordered events, that is, we want to be able to determine if, once an event (the `NULL` message) is received, there is some other event (sent tuples) 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 (12) can be modified as follows:

$$\texttt{SealR}_{snd}(i, u) \leftarrow \texttt{CntR}(u), \texttt{Id}(i), \tag{16}$$

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

where SealR is now a binary relation containing also the number of tuples originally emitted for R. Before applying (13)–(15), we have to ensure that the number of tuples over R is equal to the number of tuples originally sent. We then add to $\mathscr{T}$ the clauses:

$$\texttt{CntR}(m\_count < \bar{u} >) \leftarrow R(\bar{u}), \tag{17}$$

$$\texttt{SmNR}(m\_sum < u >) \leftarrow \texttt{SealR}(i, u), \tag{18}$$

counting the number of tuples in R and the total number of tuples over R derived globally, and finally we modify equation (15) as follows:

$$\texttt{FSR}() \leftarrow \texttt{CntSlR}(u), \texttt{CntAll}(u), \texttt{SmNR}(v), \texttt{CntR}(v). \tag{19}$$

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 also been received.

6.3 Coordination under arbitrary delay

If now we assume that condition **S3**″ does not hold, we are into the initial *rsync* semantics whereby delays can be arbitrary long although finite. Surprisingly, in this situation, we have that monotonic unconnected queries become coordination-free. To see why this is the case, first remark that a fact emitted at round t is still delivered at most at time $\theta(t+1) + \text{del}$, but, since del is now arbitrary, we are not assured that $\theta(t+1) \leqslant \theta(t+1) + \text{del} < \theta(t+2)$ any more. Despite this, the notion of coordination still maintains its semantics, even if, in this case, the coordination pattern may span multiple rounds. Let $\mathscr{N}$ be a specification parallelly computing a monotonic unconnected query, $\mathbf{I}$ an instance, and (N, D, H) a non-trivial configuration. Under the *rsync* semantics, we have that the system $\mathscr{S}^{\text{rsync}}_{\mathscr{N}}(N, D, H, \mathbf{I})$ is composed by multiple convergent runs, modeling the fact that sent tuples can be non-deterministically received in different rounds. A configuration (N, D, H) can then be chosen – e.g., the one where D installs the entire initial instance $\mathbf{I}$ on every node – so that no coordination pattern arises because the final state is already reached without having any broadcasted fact been received.

Example 17

Consider the monotonic unconnected query of Example 8. Assume a *rsync* specification $\mathscr{N}$ defined as in Example 8 but where now S is hashed on both attributes, U is broadcasted, and $\mathscr{Q}$ contains also the query:

$Q_{\text{out}}(u, v) \leftarrow S(u, v), T(\_)$. Consider the non-trivial configuration (N, D, H) in which D installs the full instance $\mathbf{I}$ on every node, while $\mathscr{H}(I_S) \subset N$ – i.e., the instance over S is not hashed to every node. We then have that a run $\rho \in \mathscr{S}^{\text{rsync}}_{\mathscr{N}}(N, D, H, \mathbf{I})$ exists such that every fact emitted over S at round t is received by round $t' \leqslant *$, while every fact over U that should be sent to a node in $N \setminus \mathscr{H}(I_S)$ is received in a round $t'' > *$. Clearly, we still have that the correct output is returned since I_T consists of at least one fact, and every sent S-tuple has been correctly received. The class defined by $\mathscr{N}$ is coordination-free.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

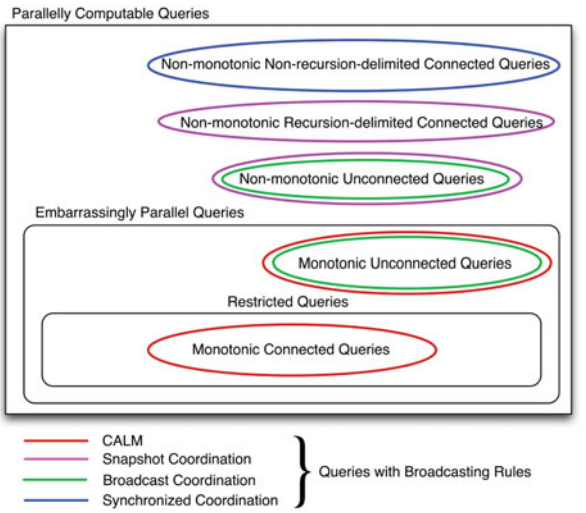

Fig. 6. Taxonomy for parallelly computable queries under *rsync*.

We can therefore conclude that the *if* direction of 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. One can also show that indeed also the *only-if* direction is satisfied. The reader can now completely appreciate how the notion of coordination we introduced perfectly aligns with the one of Ameloot *et al.* (2013) when arbitrary delay comes into play (Contribution 6): Embarrassingly, parallel queries are all coordination-free. Nevertheless, our definition is more general since it can be seamlessly used in both synchronous and asynchronous systems.

Figure 6 shows the new taxonomy when *rsync* systems are considered.

Remark: From a states of knowledge perspective, in *bsp* systems (respectively *bsp-d* – *rsync*), *common knowledge* (*δ-common knowledge*) can be obtained by simply using broadcasting (Fagin *et al.* 2003). 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 *bsp* – *bsp-d* just because of the tight requirements imposed on the system; the latter are required by the actual semantics of the query.

7 Comparison with other work

In this period, we are witnessing new trends such as *cloud computing* and *multicore processing* becoming popular. It is well-known that programming such architectures is very difficult, thus *declarative networking* has been proposed to simplify such task. The idea of declarative networking is to use high-level, declarative languages, leaving to the system the burden of organizing an efficient execution plan (Ameloot 2014).

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

In this paper, we propose to apply the same techniques to synchronous systems, in order to set forth the theoretical basis also for parallel dataflow optimizations. This application was first identified by Hellerstein (2010), who also pointed out that a tradeoff exists between efficiency of pipelining and fault-tolerance provided by full materialization and that, however, the run-time should decide which of the two strategies must be selected. Similarly, decomposable plans were identified in the 80's to speed up the evaluation of DATALOG programs through parallel execution (Wolfson and Silberschatz 1988). The general pivoting technique (Seib and Lausen 1991) implemented in BigDatalog (Shkapsky *et al.* 2016) provides only a sufficient condition to determine if a program is decomposable; decomposability is in fact undecidable in general (Wolfson and Ozeri 1990). Clearly, a relation exists between decomposable programs and programs that distribute over components: Every decomposable program distributes over components, while the opposite is not true. For instance, the following program distributes over components, but is not decomposable (Wolfson and Ozeri 1990):

$$Q(u, v) \leftarrow R(u, w), E(w, x), F(x, v),$$
$$R(u, v) \leftarrow G(u, v).$$

Ameloot *et al.* (2015) consider a superset of decomposable plans called *parallel-correct*, where queries are parameterized by a *distribution policy* and are allowed to generate not-unique facts.

In Afrati *et al.* (2011), the authors study how the MapReduce model can be extended with recursion. Additionally, a model is proposed suggesting that the optimal computation time can be obtained by minimizing the volume of data passed as input to each task. This work is orthogonal to our contribution: in fact, while we focus on how a property of queries (coordination freeness) could be used to optimize queries, (Afrati *et al.* 2011) mainly focus on adding support for recursion to a MapReduce framework. In our DATALOG implementation of Shkapsky *et al.* (2016), we also proposed a better way to support recursion in Apache Spark. From our practical experience, we found that the best way to implement transitive closure is through a decomposable plan which not necessary is optimal from a data-volume perspective since the full input dataset is passed to each task in each iteration.

The fact that CALM does not hold in general in *rsync* systems was first suggested in Wolfson and Ozeri (1990)[27] and only recently, with the advent of parallel processing systems such as MapReduce and Spark, revamped by Interlandi and Tanca (2015) for distributed parallel settings. From the latter work, we borrow the basic techniques we used to build the hashing transducer network model and our notion of coordination-freedom. Our computational model merges the original transducer network model of Ameloot *et al.* (2013) – representing how distributed computation is carried out by an asynchronous system – with the BSP model of Valiant (1990). We differ from the original transducer network model both semantically – our definition of global transition implements a synchronous and

[27] More precisely, they identified that a class of non-monotonic program exists that is communication-free.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

reliable communication model – and structurally – we have (i) an input clock driving the computation, and (ii) a special environment transducer modeling everything not functionally related with the system. Additionally, our bounded-delay and asynchronous BSP models are somehow related to the *Stale Synchronous Parallel* and A-BSP models introduced in Cui *et al.* (2014). We borrow the concept of environment from the multi-agent systems domain (Fagin *et al.* 2003). Although in contexts different than ours, synchronous transducer networks were also employed in Furche *et al.* (2014) and Interlandi *et al.* (2013). Another interesting model related to ours is the *Massive Parallel Model* of Koutris and Suciu (2011). In Massive Parallel Model, each round is divided into three phases: the usual computation and communication phases, and a broadcasting phase. As we have demonstrated, in parallel systems broadcasting implements coordination, therefore Massive Parallel Model expresses exactly those queries that require coordination in order to proceed. Koutris *et al.* showed that by employing their model, a specific class of chained conjunctive queries, denoted *tall-flat*, can be computed in one round by a load-balanced algorithm. Conversely, if a query is not tall-flat, then every algorithm consisting of one round is not load-balanced. This work as well as Ameloot *et al.* (2015) focus on how to efficiently execute queries in parallel. Conversely, our main focus is on how to extend CALM over *rsync* systems in order to unlock asynchronous plans. A similar investigation on efficiency is among our future plans.

The reader could be induced to believe that CALM indeed would not hold in synchronous settings, since systems of this kind already embed some notion of coordination. Indeed, Ben-Zvi and Moses (2010; 2011) prove that this is 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.

Our work is addressing a complementary domain with respect to *Bloom* (Alvaro *et al.* 2011, 2014) In Bloom programs, *points of order* are identified, i.e., code positions defining a non-monotonic behavior that could bring inconsistent outcomes (Alvaro *et al.* 2011). From our perspective, points of order identify where an indirect information flow exists. In Alvaro *et al.* (2014), two different coordination strategies have been identified at the basis of the cause of inconsistency: *sealing* and *ordering*. They are both comparable to our snapshot coordination. In addition, we have identified broadcasting and synchronized coordination. Finally, note that the CALM principle in its original form is satisfied only if no node is granted access to any information on how data was originally distributed; in this case, in fact, certain weaker forms of monotonic programs can be evaluated in a coordination-free way (Zinn *et al.* 2012; Ameloot *et al.* 2014). From our viewpoint, this is possible because the way data is distributed is already common knowledge before the computation starts, i.e., nodes already embed a notion of coordination. In practice, using synchronous specifications, nodes are able to compute non-monotonic queries in a coordination-free way "by construction," without any awareness of how data was initially partitioned. We are planning to investigate how the weaker forms of monotonicity identified in Ameloot *et al.* (2014) are related to our work, and whether a tradeoff exists between "distribution awareness" and "synchronization."

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

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 optimizations such as pipelining and decomposability when allowed by the program logic. This topic has recently acquired much attention because, in spite of the increasing number of applications showing better performance (and accuracy) for asynchronous execution over synchronous one (Niu *et al.* 2011; Cui *et al.* 2014; Xie *et al.* 2015), only few practical systems provide this feature as optimization (Han and Daudjee 2015; Shkapsky *et al.* 2016).

To reach our goal, we have introduced a new abstract model emulating BSP computation, and a novel interpretation of coordination with sound logical foundations in distributed knowledge reasoning. By exploiting such techniques, we have shown that the CALM principle indeed holds also in *rsync* settings, but in general only for the subclass of monotonic queries defined as connected. Finally, we have drawn attention to a hierarchy of queries with related coordination-patterns and we showed how our definition of coordination-freedom is related to the assumptions imposed on the behavior of the system: Our formalization generalizes the one employed by Ameloot *et al.* because applicable in synchronous as well as asynchronous settings.

Our next step will be to investigate to which extent the CALM principle is satisfied when queries with *aggregates* are considered; *monotonic aggregation* (Ross and Sagiv 1997; Mazuran *et al.* 2013) has been a hot topic in databases for many years: Does a relationship between monotone computation and coordination-freedom exist also for aggregate queries?

Finally, consider that all the queries in this paper are formulated in some sub-language of DATALOG$^{\neg}$. In the last few years, DATALOG^{+-} (Calì *et al.* 2011) was defined: a family of rule-based languages that extends DATALOG to capture the most common ontology languages for which query answering is tractable, and provides efficiently checkable, syntactic conditions for decidability and tractability. We plan to study extensions of our work to (sub-languages of) DATALOG^{+-}, in order to apply our results to semantic web settings.

References

Abiteboul, S., Hull, R. and Vianu, V. 1995. *Foundations of Databases.* Addison-Wesley, Boston, MA, USA.

Abiteboul, S., Vianu, V., Fordham, B. and Yesha, Y. 2000. Relational transducers for electronic commerce. *Journal of Computer and System Sciences 61,* 2, 236–269.

Afrati, F., Cosmadakis, S. S. and Yannakakis, M. 1995. On datalog vs. polynomial time. *Journal of Computer and System Sciences 51,* 2, 177–196.

Afrati, F. N., Borkar, V., Carey, M., Polyzotis, N. and Ullman, J. D. 2011. Map-reduce extensions and recursive queries. In *Proc. of the 14th International Conference on Extending Database Technology, EDBT/ICDT '11.* ACM, New York, NY, USA, 1–8.

Afrati, F. N. and Ullman, J. D. 2010. Optimizing joins in a map-reduce environment. In

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press

Proc. of the 13th International Conference on Extending Database Technology, *EDBT '10*. ACM, New York, NY, USA, 99–110.

Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.-C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M., Schelter, S., Höger, M., Tzoumas, K. and Warneke, D. 2014. The stratosphere platform for big data analytics. *International Journal on Very Large Data Bases*, *23*, 939–964.

Alvaro, P., Conway, N., Hellerstein, J. and Marczak, W. R. 2011. Consistency analysis in bloom: A calm and collected approach. In *Proc. Conference on Innovative Data Systems Research CIDR*, 249–260.

Alvaro, P., Conway, N., Hellerstein, J. M. and Maier, D. 2014. Blazes: Coordination analysis for distributed programs. In *To appear in Proc. of the IEEE 30th International Conference on Data Engineering*, *ICDE '14*. IEEE Computer Society, Washington, DC, USA.

Ameloot, T. J. 2014. Declarative networking: Recent theoretical work on coordination, correctness, and declarative semantics. *SIGMOD Record 43,* 2, 5–16.

Ameloot, T. J., Geck, G., Ketsman, B., Neven, F. and Schwentick, T. 2015. Parallel-correctness and transferability for conjunctive queries. In *Proc. of ACM Symposium on Principles of Database Systems*.

Ameloot, T. J., Ketsman, B., Neven, F. and Zinn, D. 2014. Weaker forms of monotonicity for declarative networking: A more fine-grained answer to the calm-conjecture. In *Proc. of ACM Symposium on Principles of Database Systems*. ACM, 64–75.

Ameloot, T. J., Ketsman, B., Neven, F. and Zinn, D. 2015. Datalog queries distributing over components. In *Proc. of International Conference on Database Theory ICDT*, 308–323.

Ameloot, T. J., Neven, F. and Van Den Bussche, J. 2013. Relational transducers for declarative networking. *Journal of the ACM 60,* 2, 15:1–15:38.

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A. and Zaharia, M. 2015. Spark sql: Relational data processing in spark. In *Proc. of the Special Interest Group on Management of Data SIGMOD*, 1383–1394.

Babaoğlu, O. and Marzullo, K. 1993. Consistent global states of distributed systems: fundamental concepts and mechanisms. In *Distributed systems,* 2nd ed. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 55–96.

Beeri, C., Naqvi, S. A., Ramakrishnan, R., Shmueli, O. and Tsur, S. 1987. Sets and negation in a logic data base language (ldl1). In *Proc. of ACM Symposium on Principles of Database Systems*. ACM, 21–37.

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

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

Ben-Zvi, I. and Moses, Y. 2014. Beyond lamport's happened-before: On time bounds and the ordering of events in distributed systems. *Journal of the ACM 61,* 2, 13:1–13:26.

Borkar, V., Carey, M., Grover, R., Onose, N. and Vernica, R. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. of the International Conference on Data Engineering, ICDE*, 1151–1162.

Brewer, E. A. 2000. Towards robust distributed systems. In *Proc. of the 19th Annual ACM Symposium on Principles of Distributed Computing*, *PODC '00*. ACM, New York, NY, USA, 7–.

Calì, A., Gottlob, G., Lukasiewicz, T. and Pieris, A. 2011. Datalog+-: A family of languages for ontology querying. In *Proc. of Datalog Reloaded – 1st International Workshop, Datalog 2010*, Oxford, UK, March 16–19, 2010. Revised Selected Papers, 351–368.

Chandy, K. M. and Lamport, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems 3,* 1, 63–75.

Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K. and Sears, R. 2010. Mapreduce online. In *NSDI*. USENIX Association. 313–328.

Cui, H., Cipar, J., Ho, Q., Kim, J. K., Lee, S., Kumar, A., Wei, J., Dai, W., Ganger, G. R., Gibbons, P. B., Gibson, G. A. and Xing, E. P. 2014. Exploiting bounded staleness to speed up big data analytics. In *Proc. of the USENIX Conference on USENIX Annual Technical Conference*, *USENIX ATC'14*. USENIX Association, Berkeley, CA, USA, 37–48.

Dean, J. and Ghemawat, S. 2008. Mapreduce: Simplified data processing on large clusters. *CACM 51,* 1, 107–113.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W. 2007. Dynamo: Amazon's highly available key-value store. *SIGOPS Operating Systems Review 41,* 6, 205–220.

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

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

Furche, T., Gottlob, G., Grasso, G., Guo, X., Orsi, G., Schallhart, C. and Wang, C. 2014. DIADEM: Thousands of websites to a single database. *PVLDB 7,* 14, 1845–1856.

Gaifman, H., Mairson, H., Sagiv, Y. and Vardi, M. Y. 1993. Undecidable optimization problems for database logic programs. *Journal of the ACM 40,* 3, 683–713.

Guessarian, I. 1990. Deciding boundedness for uniformly connected datalog programs. In *International Conference on Database Theory ICDT*, Vol. 470, S. Abiteboul and P. Kanellakis, Eds. Lecture Notes in Computer Science, 395–405.

Han, M. and Daudjee, K. 2015. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *VLDB 8,* 9, 950–961.

Hellerstein, J. M. 2010. The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Record 39*, 5–19.

Hunt, P., Konar, M., Junqueira, F. P. and Reed, B. 2010. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. of the USENIX Conference on USENIX Annual Technical Conference*, *USENIXATC'10*. USENIX Association, Berkeley, CA, USA, 11–11.

Interlandi, M. and Tanca, L. 2015. On the CALM principle for BSP computation. In *Proc. of the Alberto Mendelzon International Workshop on Foundations of Data Management*.

Interlandi, M. and Tanca, L. 2017. On the CALM principle for bulk synchronous parallel computation. *CoRR abs/1405.7264*.

Interlandi, M., Tanca, L. and Bergamaschi, S. 2013. Datalog in time and space, synchronously. In *Proc. of the Alberto Mendelzon International Workshop on Foundations of Data Management*.

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

Koutris, P. and Suciu, D. 2011. Parallel evaluation of conjunctive queries. In *Proc. of ACM Symposium on Principles of Database Systems*, *PODS '11*. ACM, New York, NY, USA, 223–234.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7, 558–565.

Lamport, L. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems 6,* 2, 254–280.

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

Mazuran, M., Serra, E. and Zaniolo, C. 2013. Extending the power of datalog recursion. *VLDBJ 22,* 4, 471–493.

Mumick, I. and Shmueli, O. 1995. How expressive is stratified aggregation? *Annals of Mathematics and Artificial Intelligence 15,* 3–4, 407–435.

Niu, F., Recht, B., Re, C. and Wright, S. J. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proc. of the 24th International Conference on Neural Information Processing Systems*, *NIPS'11*. Curran Associates Inc., USA, 693–701.

Olston, C., Reed, B., Srivastava, U., Kumar, R. and Tomkins, A. 2008. Pig latin: A not-so-foreign language for data processing. In *Proc. of the Special Interest Group on Management of Data, SIGMOD*. ACM, 1099–1110.

Ramakrishnan, R., Beeri, C. and Krishnamurthy, R. 1988. Optimizing existential datalog queries. In *Proc. of Symposium on Principles of Database Systems*. ACM, 89–102.

Ramakrishnan, R. and Ullman, J. D. 1995. A survey of deductive database systems. *Journal of Logical Programming 23,* 2, 125–149.

Ross, K. A. and Sagiv, Y. 1997. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences 54,* 1, 79–97.

Seib, J. and Lausen, G. 1991. Parallelizing datalog programs by generalized pivoting. In *Proc. of ACM Symposium on Principles of Database Systems*, 241–251.

Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T. and Zaniolo, C. 2016. Big data analytics with datalog queries on spark. In *Proc. of the International Conference on Management of Data*, *SIGMOD '16*. ACM, New York, NY, USA, 1135–1149.

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

Valiant, L. G. 1990. A bridging model for parallel computation. *CACM 33,* 8, 103–111.

Vogels, W. 2009. Eventually consistent. *Communications of the ACM 52,* 1, 40–44.

Wolfson, O. and Ozeri, A. 1990. A new paradigm for parallel and distributed rule-processing. In *Proc. of the Special Interest Group on Management of Data SIGMOD*, 133–142.

Wolfson, O. and Silberschatz, A. 1988. Distributed processing of logic programs. In *Proc. of the Special Interest Group on Management of Data SIGMOD*, 329–336.

Xie, C., Chen, R., Guan, H., Zang, B. and Chen, H. 2015. Sync or async: Time to fuse for distributed graph-parallel computation. In *Proc. of the Principles and Practice of Parallel Programming PPoPP*, 194–204.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. and Stoica, I. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the NSDI*.

Zaniolo, C., Yang, M., Das, A. and Interlandi, M. 2016. The magic of pushing extrema into recursion: Simple, powerful datalog programs. In *Proc. of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management*, Panama City, Panama, May 8–10, 2016.

Zaniolo, C., Yang, M., Das, A., Shkapsky, A., Condie, T. and Interlandi, M. 2017. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *TPLP 17,* 5–6, 1048–1065.

Zaniolo, C., Yang, M., Interlandi, M., Das, A., Shkapsky, A. and Condie, T. 2018. Declarative bigdata algorithms via aggregates and relational database dependencies. In *Proc. of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management*, Cali, Colombia, May 21–25, 2018.

Zinn, D., Green, T. J. and Ludäscher, B. 2012. Win-move is coordination-free (sometimes). In *International Conference on Database Theory, ICDT*. ACM, 99–113.

https://doi.org/10.1017/S147106841800042X Published online by Cambridge University Press