Declarative Networking: Recent Theoretical Work on Coordination, Correctness, and Declarative Semantics*

Tom J. Ameloot†
Hasselt University &
Transnational University of Limburg
Diepenbeek, Belgium
tom.ameloot@uhasselt.be

ABSTRACT

We discuss recent theoretical results on declarative networking, in particular regarding the topics of coordination, correctness, and declarative semantics.

1. INTRODUCTION

Cloud computing refers to the principle that computations are distributed over a network of computing nodes to increase parallelism [54]. Cloud computing is challenging to implement because messages can be delayed, computing nodes can crash, network links can be broken, etc. It seems desirable to abstract away from some of these technical aspects, and let them be automatically handled by a suitable framework.

Paradigms for cloud computing have emerged that hide some technical aspects and provide intuitive concepts instead. Well-known examples are MapReduce [24], Pregel [44], and GraphLab [41]. These paradigms suggest concrete and yet simple ways to think about cloud computing. The programmer typically provides the functionality in the form of a few modules, and the runtime engine takes care of the actual distributed execution of these modules. In general, the modules are specified with imperative programming languages.

Now, declarative networking is another proposal to simplify programming of cloud computing, using high-level declarative languages instead of imperative languages. The programmer expresses what should happen instead of how to effectively achieve this. The runtime engine will generate a distributed physical query plan to perform the desired cloud computation. For example, regarding messages, a declarative program will only generate the event to send a specific message to a certain recipient, and the runtime engine chooses some efficient delivery strategy. Languages for declarative networking elegantly combine messaging features with local computation. Originally, the term declarative networking referred specifically to Datalog-inspired languages, and also more specifically to networking protocols [39]. In the meantime, the languages remain mostly Datalog-inspired, but several works now also consider their use for general distributed database queries. In this context, cloud data is typically viewed as a distributed database.

In this paper, we discuss recent theoretical results on declarative networking, thereby complementing the surveys of Hellerstein [30] and Loo et al. [40] that discuss various applications and practical aspects of declarative networking.

We give an overview of the paper. First, Section 2 briefly introduces basic database and Datalog terminology. The following two sections discuss theoretical results on distributed execution. Section 3 discusses coordination, and studies in particular the CALM conjecture by Hellerstein. Section 4 discusses correctness of distributed computations, including decidability results. Next, Section 5 highlights features of languages in declarative networking and reviews declarative semantics for such languages; in this context, we also examine a second conjecture by Hellerstein, namely, the CRON conjecture. Section 6 provides directions for further work.

2. PRELIMINARIES

The purpose of this section is to introduce some concepts that are frequently used in this paper [3].

*Database Principles Column. Column editor: Pablo Barceló. Department of Computer Science, University of Chile, Santiago, Chile. E-mail: pbarcelo@dcc.uchile.cl

†PhD Fellow of the Fund for Scientific Research, Flanders (FWO).

A *database schema* $\mathcal{D}$ is a set of pairs (R, k), where R is a relation name and $k \in \mathbb{N}$ is its associated arity. A *fact* over $\mathcal{D}$ is of the form $R(\bar{a})$ where R is a relation name from the schema and $\bar{a}$ is a tuple of values matching the arity of the relation. An *atom* over $\mathcal{D}$ is of the form $R(\bar{u})$ where R is again a relation from the schema, and $\bar{u}$ is now a possibly mixed tuple of values and variables, matching the arity of the relation.

A *conjunctive query with negation* over $\mathcal{D}$ is of the following form:

$$T(\bar{u}) \leftarrow R_1(\overline{v_1}), \ldots, R_p(\overline{v_p}), \neg S_1(\overline{w_1}), \ldots, \neg S_q(\overline{w_q}).$$

where $T(\bar{u})$ and all $R_i(\overline{v_i})$ and $S_j(\overline{w_j})$ are atoms over $\mathcal{D}$. A conjunctive query with negation may also be called a (Datalog) *rule*. Atom $T(\bar{u})$ is called the *head* and the other atoms constitute the *body*. The order of body atoms is usually irrelevant. The R_i-atoms are called *positive*: they test the presence of facts. The S_j-atoms are called *negative*: they test the absence of facts; the symbol '$\neg$' stands for negation. For simplicity, we make the common assumption that all variables of a rule occur in its positive body atoms.

To evaluate a rule on a set of input facts, we seek a substitution of the rule variables by values so that facts resulting from positive body atoms occur in the input and facts resulting from negative body atoms do not occur in the input. Applying this substitution to the head atom results in a fact, the so-called *derived* fact.

A Datalog program over a database schema is a set of rules over this schema. A Datalog program is called *positive* when its rules contain only positive body atoms.[1] A Datalog program is called *recursive* when some head relations of rules also occur in rule bodies.

3. COORDINATION

Coordination means that nodes of a cloud are trying to obtain a global consensus. For example, by exchanging messages about the presence or absence of data in the cloud, coordination protocols could ensure that all nodes have the desired data before applying negation in their local computation. Because computation at all nodes is halted during coordination, which reduces parallelism, we want to avoid coordination as much as possible.

Recent research on coordination consists of two main approaches. Both approaches provide indications about how efficient the distributed runtime engines for declarative networking can be made. The first approach, as embodied by the CALM conjecture, investigates which distributed computations can completely avoid coordination and are thus "embarrassingly parallel" [30]. For distributed computations that can not completely avoid coordination, the other approach quantifies the required amount of coordination. Even for computations that can avoid coordination, a quantitative approach can shed light on the costs involved. These approaches are complementary, and are discussed in Sections 3.1 and 3.2 respectively.

3.1 CALM Conjecture

During his PODS 2010 keynote, Hellerstein presented a number of intriguing conjectures to the database community [30]. The first conjecture is called the CALM conjecture (Consistency And Logical Monotonicity), that we repeat for convenience:

CONJECTURE 1 (CALM [30]). *A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog.*

We explain the meaning of this conjecture. Eventual consistency is a correctness notion: it indicates that the program can tolerate arbitrary message delays, as occurring in asynchronous communication settings (cf. Section 4). Coordination-freeness means that coordination is completely avoided. Monotonic Datalog refers to positive Datalog, that is indeed restricted to so-called *monotone* computations where previous output facts remain valid whenever the input is extended with new facts (and new output facts may also be produced). So, the CALM conjecture suggests that a distributed program can avoid coordination (and stay correct) if and only if that program is expressible in positive Datalog.

One direction of the CALM conjecture was already known: positive Datalog programs can be implemented without coordination [39]. This actually holds more generally for all monotone programs, even those that are not expressible in positive Datalog: the main intuition here, is that the nodes can send the input data to each other and steadily accumulate these messages; whenever a new message arrives, a node can always again evaluate the monotone program. Because the program is monotone, no wrong outputs are produced by previous evaluations. This strategy results in eventual consistency.

But the other direction of the CALM conjecture appears new: it suggests an upper bound on the ex-

[1] The term "Datalog" originally denoted programs with only positive bodies [3]. But to simplify terminology, this paper also uses the term for programs with negative body atoms.

pressivity of distributed programs that can proceed without coordination. This direction has prompted several investigations, that we discuss below.

Coordination-freeness and monotonicity. Because the CALM conjecture was only stated informally, it had to be formalized first. Ameloot et al. [12] have proposed a formal definition of coordination-freeness: a program is called coordination-free when for each set of input facts, there is some right way to distribute these facts over the network so that the nodes can already compute the entire output without communicating. Intuitively, the program still has to be correct for all possible input distributions, but there is a right distribution enabling an embarrassingly parallel execution. Although the original CALM conjecture mentioning Datalog was disproved in the formal framework of Ameloot et al., nonetheless the main intuition of the conjecture turns out to hold [12]: a distributed program is coordination-free if and only if it is monotone.

Coordination-freeness and non-monotonicity. Using the same definition of coordination-freeness as Ameloot et al. [12], Zinn et al. [55] have subsequently obtained additional insights on the CALM conjecture. Surprisingly, it turns out that some *non-monotone* programs are coordination-free when each node is given knowledge about the *distribution policy* of the global input data. This way, a node can sometimes locally conclude that certain input facts are globally absent, allowing some non-monotone programs to proceed without coordination. In the previous model [12], where this policy is not exposed, a node would always have to *coordinate* with all other nodes to conclude such global absences. Moreover, it turns out that in some variations of the model considered by Zinn et al., all programs can be made coordination-free; these variations, however, are quite expensive in terms of how much additional data each node should have.

Weaker forms of monotonicity. Recently, the results by Ameloot et al. [12] and Zinn et al. [55] have been combined in a more unified theory on the CALM conjecture [11]. In particular, the non-monotone programs that can avoid coordination, as identified by Zinn et al. [55], have been characterized semantically with weaker forms of monotonicity: two classes have been identified, called *domain-distinct-monotone* and *domain-disjoint-monotone* programs, that we explain below.

Recall that for an "ordinary" monotone program, previous output facts remain valid whenever the input is extended with arbitrary new facts (and new output facts may also be produced). Now, a program is called domain-distinct-monotone if previous output facts remain valid when we extend the input with facts that each contains at least one new value not yet occurring in the old input. For example, the difference $R \setminus S$ of two unary relations R and S is domain-distinct-monotone: indeed, new input facts added to R will not shrink the output, and new facts added to S will not subtract from R if they contain at least one value not yet occurring in the old input. Note that $R \setminus S$ is not monotone.

A program is called domain-disjoint-monotone if previous output facts remain valid when we extend the input with facts that share no values with the old input. For example, the complement of the transitive closure on a binary edge relation R is domain-disjoint-monotone: new edges that share no values with the previous input can not establish a path between old vertices. This program is not domain-distinct-monotone.

Note that ordinary monotonicity implies domain-distinct-monotonicity and that the latter implies domain-disjoint-monotonicity.

In accordance with the results of Zinn et al. [55], domain-distinct-monotone programs can be implemented without coordination if the nodes of the cloud are made aware of how input facts are distributed. The same awareness is needed for domain-disjoint-monotone programs to be implemented without coordination, but also with the additional assumption that nodes are now "responsible" for input *values*: each node is initialized with all input facts containing any value the node is responsible for.

Efficient coordination-free strategies. The above works theoretically relate distributed coordination to program monotonicity. They are complemented by works that investigate efficient implementation strategies for coordination-free programs. For example, the works of Loo et al. [38, 39] and Nigam et al. [47] provide concrete algorithms for the case of distributed positive Datalog programs. Whenever some input facts change, these algorithms efficiently update the state at the nodes of a cloud. To avoid recomputing the entire state at every node, only incremental changes are propagated. This reduces communication and needless recomputation. These algorithms are coordination-free; handle recursive Datalog; and, tolerate messages delayed by the network, i.e., they are eventually consistent.

3.2 Quantification

The second approach to understanding coordination is to quantify the amount of coordination, and any related costs.

First, Alvaro et al. [7, 8] propose program analysis techniques to detect code fragments where coordination is perhaps overused. This way, some uses of coordination could be replaced with strategies like eventual consistency, reducing the overall amount of coordination.

Koutris and Suciu [33] define the massively parallel model of computation (MP). An execution in this model is a sequence of global MP steps. In each step, the nodes first communicate and then they perform local computation. Each step represents a global round of coordination. Koutris and Suciu also define when an algorithm is load-balanced in this model: this intuitively means that each server locally processes an equal share of the total problem size. In this setting, Koutris and Suciu prove that the *tall-flat conjunctive queries* are precisely those conjunctive queries that can be computed in one MP step in a load-balanced way.

Beame et al. [18] extend the work of Koutris and Suciu by considering a parameter to control the amount of data that each node may receive during a step. Higher values of the parameter allow more replication of data. Less replication is viewed as more efficient. Beame et al. quantify the replication required when a computation may only use one global communication step. Beame et al. also quantify the number of global steps required when the allowed replication is fixed.

Interestingly, the positive conjunctive queries considered for load-balanced algorithms [33] and replication [18] can be implemented with a coordination-free strategy in the models used for the CALM conjecture [12, 55]. However, these coordination-free strategies would not be efficient because they gradually replicate the input over the network. Thus, the notion of coordination-freeness is only part of a larger picture, where costs can be formalized and measured in multiple ways.

4. CORRECTNESS

Cloud computing often works over an asynchronous communication model, where messages can be arbitrarily delayed. Larger networks are typical settings with asynchronous communication, because routers can forward messages differently depending on network congestion, subjecting messages to unpredictable latencies.

It is important to design distributed programs that tolerate message delays. We may call a distributed program correct if, for each input, it succeeds in producing the desired output no matter how much messages are delayed. We discuss two main strategies for ensuring correctness, namely, construction and verification, given in Sections 4.1 and 4.2 respectively.

4.1 Constructive Approach

A first main strategy to obtain correct distributed programs, is to use certain principles for program construction.

On one side of this spectrum, we have eventual consistency [52, 30, 16]. This means that the output is eventually produced if messages are eventually delivered, in some arbitrary fashion. There is no coordination here. It is well-known that monotone computations can be executed in an eventually consistent way: indeed, whenever a node receives a new message, it can simply recompute the local result, which is guaranteed to be part of the overall output by monotonicity (cf. Section 3.1). Another approach to eventual consistency consists of so-called commutative replicated data types, where messages represent commutative operations, that are thus resilient to unpredictable reorderings [50, 22, 16].

Coordination protocols are at the other side of the spectrum, e.g., used when the computation is not monotone or if messages do not commute. Note, however, that some classes of non-monotone computations can be implemented without coordination [55].

4.2 Deciding Correctness

A second main strategy, is to decide correctness for distributed programs.

Ameloot and Van den Bussche [13] have investigated decidability of correctness for distributed programs in which each computing node of a cloud is represented by a (relational) transducer [4, 25, 26, 27, 51]; such a distributed program is referred to as a transducer network. Here, a transducer is a collection of queries over a database schema, where each of the relations is used for either input, output, memory, messages, or auxiliary system relations; the queries update the output and memory relations, and generate messages.

Now, Ameloot and Van den Bussche [13] define correctness as a confluence notion: a distributed program is called confluent if for any two finite execution traces on the same input, the second trace can always be extended to obtain the (partial) output of the first trace. Intuitively, the prior execution of the program will not prevent outputs from being produced. The opposite of confluence is called

diffluence. Deciding diffluence for so-called *simple* transducer networks, where transducers are implemented with restricted conjunctive queries, turns out to be NEXPTIME-complete. The restrictions of simple transducer networks are: *(i)* the network is recursion-free, where rules cannot be mutually recursive through positive body atoms; *(ii)* deleting from output and memory relations is forbidden; *(iii)* negation on message relations is forbidden; *(iv)* rules inserting into output and memory relations must be "message-bounded";[2] finally, *(v)* message-sending rules only use input and message relations.

Ameloot and Van den Bussche [13] have shown that simple transducer networks compute exactly all distributed queries expressible by unions of conjunctive queries with negation, or equivalently, the existential fragment of first-order logic. Compared to standard database queries, the location of facts matters for distributed queries. We may conclude that simple transducer networks are indeed a weaker computational model, but that is not totally useless.

Ameloot [10] has investigated decidability of a second formalization of correctness, referred to as consistency, also appearing in prior works [2, 12]: a distributed program is called consistent if any two infinite fair execution traces on the same input yield the same output. The fairness conditions demand that all sent messages are eventually delivered and that all nodes are made active infinitely often. Deciding inconsistency for simple transducer networks is again NEXPTIME-complete. The expressivity of simple transducer networks is the same under both confluence and consistency.

5. DECLARATIVE LANGUAGES

This section is devoted to languages in declarative networking, and their semantics. Section 5.1 highlights some important features of languages in declarative networking. Section 5.2 discusses operational semantics. Section 5.3 discusses declarative semantics as an alternative to operational semantics; we also use this context to discuss a second conjecture by Hellerstein, namely, the CRON conjecture [30].

5.1 Datalog Variants

As we have mentioned in the Introduction, declarative networking originally developed around Datalog [39, 30]. Today, Datalog is still an attractive foundation for declarative networking, because it allows expressing advanced algorithms with relatively few lines of code [30]. We are also seeing a more general interest in Datalog [23, 31, 17].

A notable language proposed in declarative networking is Dedalus [9, 30], a minimalistic extension of Datalog to the distributed setting: it only provides basic features for reasoning about distributed facts, and it provides a simple way to designate some facts as messages between nodes. Initial expressivity and complexity properties of Dedalus are provided by Ameloot and Van den Bussche [14].

Dedalus [9] and its predecessor languages [39] have influenced other recent language designs in declarative networking such as WebdamLog [2, 1], Bloom [7, 8], and several other works [29, 32, 37].

Location specifiers. A frequently occurring feature in declarative networking, is the use of *location specifiers* to tag facts with the node that stores that fact [39]. Accordingly, rules have additional variables for location specifiers in head and body atoms; each atom contains precisely one such variable. Often, the same location specifier is used in all body atoms, meaning that the rule can be evaluated locally on a single node. Now, if the head location specifier variable is different from the body location specifier variable, derived facts are sent as messages to the node indicated by the head variable. Otherwise, derived facts are stored locally. For each case, the runtime engine handles the details of message sending or local storage, respectively.

Delegation. The language WebdamLog [2, 1] has introduced the novel feature to *delegate* at runtime a piece of functionality, as represented by a set of rules, to the node with the best opportunity to evaluate these rules; this typically means that the node has the required data. Also, an important design principle of WebdamLog is that previously unseen nodes can start to participate in a computation that is already running, each contributing new local rules. To make different WebdamLog rules still globally interoperate, a programmer could write variables in place of relation names.

Time. In its "unsugared" presentation, Dedalus explicitly exposes *time variables* in its rules. The intention of exposing time, is to more clearly reason about dynamic changes to the memory of the computing nodes; all from within the declarative program itself, instead of deferring this aspect to the runtime engine. Concrete values for time variables may be called *timestamps*, and are often just natural numbers. The exposure of time connects Dedalus to temporal deductive databases and tem-

[2]This corresponds to *input-boundedness*, as first identified by Spielmann [51] and further investigated by Deutsch et al. [26, 27].

poral logic programming (cf. Section 5.3.2).

5.2 Operational Semantics

To describe how programs in declarative networking are distributedly executed, often an *operational semantics* is used. This represents how the runtime engine underneath the declarative language works. The results on coordination (Section 3) and correctness (Section 4) are about such operational semantics.

It is well understood how such an operational semantics might be defined [27, 46, 29, 2, 12]. Typically, a transition system is used, describing how the cloud moves from one global state to another global state as the result of local computation at nodes and message sending between nodes. This transition system is infinite because nodes can run indefinitely and keep sending messages so that an unbounded number of messages can be floating around in the network. In addition, the transition system is highly nondeterministic, because each transition chooses which node becomes active and which messages are delivered. This allows representing asynchronous communication, where messages can be delayed and eventually be delivered out of order.

Ameloot et al. [12] have defined an operational model for declarative networking where each computing node is implemented with a local relational transducer (cf. Section 4.2). Fragments of Datalog may be used to implement such transducers, for example, unions of conjunctive queries with negation or first order logic. Ameloot et al. [12] also provide expressivity results in this operational model. For example, non-monotone distributed computations require each node to have access to its own identifier and the identifiers of all the other nodes. Also, the transducer model turns out to be quite natural: it only introduces a kind of iteration to the local query language of the transducers.

Because an operational semantics might become difficult for a programmer to imagine, it is useful to look at suitable abstractions. This may be called a *declarative semantics*, which is discussed next.

5.3 Declarative Semantics

By hiding technical details of operational executions, a declarative semantics can help separate the meaning of a program from the actual distributed execution strategies. This way, old distributed execution strategies can be replaced with new strategies, as long as the new strategies satisfy the same declarative semantics.

Based on their Datalog origin, languages in declarative networking have already explored some well-known semantics of Datalog as candidates for their own declarative semantics: Section 5.3.1 discusses simple fixpoint semantics; Section 5.3.2 discusses syntactically and temporally stratified semantics; and, Section 5.3.3 discusses the stable model semantics. Although the stable model semantics might be less intuitive for the programmer, it provides avenues for new theoretical and practical research. In particular, Section 5.3.4 discusses how stable models allow reasoning about message causality.

5.3.1 Simple Fixpoint Semantics

Although strictly speaking it is still an operational semantics, a fixpoint semantics can be an intuitive semantics for declarative networking. Essentially, this semantics transforms an initial set of input facts by successively applying updates generated by triggered rules. Updates could be insertions, or deletions when rule heads contain negation. The computation ends when no more facts can be added or removed, i.e., when a fixpoint has been reached. Sometimes no fixpoint is reached.

In declarative networking, the programmer can imagine that the fixpoint semantics is applied to a centralized, holistic Datalog-like program having access to all data and rules in the cloud. Here, communication is viewed as happening instantaneously, that is, asynchronous communication is abstracted away. It is important, of course, to prove that output under this fixpoint semantics really corresponds to the output produced by the distributed execution.

Positive programs. For positive Datalog-like languages, i.e., programs not using negation (and not doing deletions), several works establish a connection between a centralized fixpoint semantics and a distributed execution [38, 39, 2, 47]. Here, the fixpoint always exists.[3] The intuition for monotone programs applies (cf. Section 3.1): using mild syntactic assumptions [2], positive programs will steadily accumulate any received messages, thereby creating the opportunity for delayed facts to still participate in the monotone computation, and relational joins in particular.

Semi-monotone programs. Zinn et al. [55] have investigated a Datalog variant called *semi-monotone*. Besides allowing rules to trigger fact insertions and deletions, this language only allows two kinds of computed relations: *(i)* relations only tested positively in rules and only inserted into, and *(ii)* rela-

[3]The fixpoint semantics for positive Datalog programs corresponds to their minimal model semantics [3].

tions only tested negatively in rules and only deleted from. For this language, Zinn et al. prove that a deterministic fixpoint semantics corresponds to distributed executions that are eventually consistent. Again, the deterministic semantics appears more intuitive compared to the nondeterministic distributed execution.

5.3.2 Stratified Semantics

In declarative networking, two variants of stratified programs have been studied.

Syntactically stratified programs. A Datalog program is *syntactically stratified* when its rules can be divided into sets, called strata, that are ordered in such a way that rule bodies apply negation only to relations computed in previous strata [3]. So, there is no recursion through negation. The program is evaluated by successively evaluating the strata: we start with the first stratum, then the next stratum, etc. Each stratum itself is evaluated under the fixpoint semantics, and it may read the facts generated by the previous stratum.

Syntactic stratification can also be defined for languages in declarative networking, e.g., for fragments of Dedalus [9] or WebdamLog [2]. The connection between a centralized semantics (cf. Section 5.3.1) and distributed executions can also be established for syntactically stratified programs, but this is more challenging compared to positive programs [30]. Indeed, a relation T could be partially computed by multiple nodes in the distributed setting. So, whenever a node x wants to apply negation to relation T (as computed in a previous stratum), node x needs to communicate with the other nodes before it can determine the presence or absence of some T-facts. One way to achieve this, is to let the global computation proceed in rounds: each round corresponds to one stratum of the centralized program, and each round is followed by a coordination phase to make sure that nodes have their required facts from the previous stratum; then the next round begins and nodes can safely apply local negation. Here, although the centralized stratified semantics is still intuitive for the programmer, the distributed execution may need to employ some expensive coordination mechanisms.[4]

Temporally stratified programs. A Datalog program is called *locally stratified* when for each input, all possible ground rules based on the input values can be grouped into strata such that ground atoms appearing negatively in rule bodies can be rule-heads only in lower strata [15].[5] Intuitively, no ground atom can negatively depend on itself. So, this condition is very much like syntactic stratification, except we now use ground rules.

A particular kind of local stratification is *temporal stratification*, that is well-studied in temporal deductive databases and temporal logic programming [53, 34, 42]. Seminal work in this field is by Chomicki and Imieliński [21, 20]. In this setting, all facts are tagged with an additional timestamp, to indicate the discrete moment on which the fact exists. Accordingly, rules will mention an additional timestamp variable for each head and body atom. Intuitively, a program is said to be temporally stratified when each head timestamp variable always represents a larger timestamp value than all timestamp variables in the accompanying rule body. This ensures that facts are only derived in the future. Negation is thus only applied to relations computed in the past, preventing cyclic dependencies involving negation through time. Besides using a model-based semantics [15], we can imagine that the program evolves from timestamp to timestamp, where facts available at the current timestamp may contribute to deriving facts at future timestamps.

In the context of declarative networking, Dedalus without (asynchronous) message rules is temporally stratified [9, 30]. In the remaining rules, all body atoms use the same timestamp variable, and the head timestamp variable is either *(i)* the same as the body timestamp variable, or *(ii)* it is restricted to be the successor of the body timestamp variable. Dedalus assumes that rules of the first kind, called deductive rules, are syntactically stratified. This ensures temporal stratification for the language without message rules.

In a similar vein, Interlandi et al. [32] give a Dedalus-inspired language for *synchronous* systems. Here, nodes of the network proceed in rounds and messages are not arbitrarily delayed. During each round, nodes share the same global clock. Interlandi et al. show that an operational semantics for their language coincides with a declarative model-based semantics of a single holistic Datalog program; this declarative semantics is enabled by the temporal stratification.

More generally, it seems that when ignoring message rules, many languages used in declarative networking can be (strictly) embedded in some of the prior languages [53, 34, 42], because the remaining rules represent local computation, which typically

[4] Hellerstein [30] makes initial suggestions to reduce the coordination complexity of stratified programs.

[5] Ground rules are obtained from original program rules by replacing their variables with concrete values.

only deals with the current time and the next time.

As an intermediate conclusion, the previous works mentioning temporal stratification for declarative networking have not yet investigated asynchronous communication, where arrival timestamps of messages can be arbitrarily into the future. Asynchronous communication can, however, be represented in detail by the stable model semantics, as discussed next.

5.3.3 Stable Model Semantics

We now discuss uses of the stable model semantics [28] for languages in declarative networking. Although it is perhaps less intuitive for the programmer, this semantics provides an interesting framework for theoretical and practical research.

Dedalus stable models. Stable models have been proposed as a way of thinking about the semantics of Dedalus programs [45]. The main idea is that, for a given distributed input, each stable model represents another way in which nondeterminism is caused by asynchronous communication.

A formal proof was provided to show the correspondence between the stable model semantics and an operational semantics [5, 6].[6] In this proof, a Dedalus program is first translated to a pure Datalog program (with negation), to which the stable model semantics is applied. This Datalog program represents the computation of the entire cloud, thus providing a kind of centralized semantics as in Section 5.3.1. The pure Datalog program gives each relation two dedicated components: one component for the location of facts and the other component for the local timestamp of facts at their location (cf. Section 5.1). Now, asynchronous communication is modeled with the choice construct by Saccà and Zaniolo [49], allowing to nondeterministically select an arrival timestamp at the addressee for each message. The pure Datalog program is also extended with auxiliary rules to enforce natural properties occurring in an operational semantics. The first natural property is *causality*: messages are only delivered in the future, and not in the past. We elaborate on this property in Section 5.3.4. The second natural property is that only a finite number of messages arrive at each timestamp of a node.

Practical answer set programming. Stable model semantics also enables verification and testing. For example, Lobo et al. [37] provide a semantics for a Dedalus-like language based on answer set programming (ASP), i.e., stable models. This is again done by translating an original program into a pure Datalog program that holistically describes the distributed computation, resembling the translation of Dedalus programs to Datalog mentioned above [5, 6]. To enforce execution properties in their semantics, like causality, Lobo et al. also specify auxiliary rules in the syntactical translation. By varying certain rules, the communication semantics can be specified, for example, whether messages are delivered synchronously or asynchronously.

By giving this holistic Datalog program to available ASP solvers, the original distributed program can be simulated and thus analyzed [37]. Under asynchronous communication, nondeterminism can occur: multiple answer sets exist, each representing a different execution of the distributed program. One might, for example, verify whether the distributed algorithm is correct in the sense of Section 4 by enumerating or sampling the answer sets. However, enumerating all possible answer sets under asynchronous communication poses scalability issues [37].

The work of Lobo et al. is extended by Ma et al. [43], who also formalize an operational semantics of distributed systems. Global properties of the system can again be analyzed by translating it into a logic program, to which an ASP solver can be applied.

In this context we can also mention an area of artificial intelligence closely related to declarative networking: programming multi-agent systems in declarative languages. The knowledge of an agent can be expressed by a logic program and agents update their knowledge by modifying their rules [36, 48, 35]. The semantics of such dynamic agents is often given by a stable model semantics, implemented in practice with ASP solvers.

5.3.4 The CRON Conjecture

The term *causality* means that an effect can only happen after its cause. In the distributed setting, this implies that a message can only arrive after it was sent. To illustrate, if node x at local timestamp 2 sends a message A that arrives at local timestamp 1 of node y, then any message B that node y sends at local timestamp 1 or later may not arrive on x at local timestamp 2 or less; put differently, local timestamp 2 of x lies in the past with respect to local timestamp 1 of y.

In practice, causality seems to be satisfied in general, except in situations like crash recovery: there,

[6] To the best of our knowledge, this is the only work to rigorously establish a connection between stable models and an operational semantics of the form discussed in Section 5.2.

an arriving or logged message could appear to come from the future when it is put side-by-side with an old state snapshot. To illustrate, suppose in our above example that x sets a local flag when it has sent message A to y and that y sends B as a reply when receiving A. If x crashes, x could now be reverted to a state without the local flag; when receiving message B in this state, node x will not expect B, as if B is coming from the future in the viewpoint of x. We may call this *non-causality*.

In this context, we can now study a second conjecture by Hellerstein, namely the CRON conjecture (Causality Required Only for Non-monotonicity):

CONJECTURE 2 (CRON [30]).
Program semantics require causal message ordering if and only if the messages participate in non-monotonic derivations.

The CRON conjecture relates causality of messages to the nature of the computations in which those messages participate, for example monotone versus non-monotone. In particular, the conjecture suggests that the order of messages (causality) is only important for non-monotone computations. Perhaps more generally, the CRON conjecture asks us to think about the need for time, like when temporal delay on messages is needed for expressivity.

One way to investigate this conjecture is as follows. As explained in Section 5.3.3, the stable model semantics for languages in declarative networking allows representing asynchronous communication in detail. In particular, the choice construct [49] allows us to nondeterministically select an arrival timestamp at the addressee for each message. But auxiliary rules were be added to enforce causality [5, 37, 6]. Now, to formalize the CRON conjecture in the Dedalus setting, Ameloot and Van den Bussche [14] study the effect of *omitting* such auxiliary rules, to see how the behavior of the program changes as a result. Concretely, omitting the auxiliary rules allows certain stable models where messages are sent "into the past", representing non-causality. Then, a Dedalus program that is already correct in an operational semantics (cf. Section 4) is said to *tolerate non-causality* when these non-causal stable models yield the same output as the operational executions.[7] In this setting, the CRON conjecture can be seen as suggesting a link between tolerance to non-causality and monotonicity, much like the CALM conjecture relates a semantic property (coordination-freeness) to monotonicity.

[7]Inside a stable model, the output at a node x is the set of all facts $\boldsymbol{f}$ for which there is a local timestamp of x so that $\boldsymbol{f}$ is present at x in all the following local timestamps.

However, Ameloot and Van den Bussche [14] show that the CRON conjecture does not hold when formalized purely semantically, where a Dedalus program is seen as expressing a database query. More concretely, both directions of the following formal conjecture can be disproved: *A Dedalus program computes a monotone query if and only if it tolerates non-causality.* However, on a more syntactical level, it can be shown that all positive Dedalus programs tolerate non-causality.[8] This result establishes a class of programs that no not require causality to be maintained on messages, for example during crash recovery.

Outside crash recovery, the extreme non-causality of sending messages "into the past" is unlikely to occur. Yet, the result that positive Dedalus programs tolerate non-causality seems distantly related to the result that positive programs have eventually consistent distributed execution strategies (cf. Section 4.1): indeed, eventually consistent programs can deal with unpredictable message reorderings, as occurring under asynchronous communication.

6. FURTHER WORK

We provide directions for further work.

Coordination.

For the CALM conjecture, we might need additional formal definitions of coordination-freeness. A problem with the previous definition [12] is perhaps that a distributed program is already coordination-free when there is one right distribution of the input on which nodes can locally compute the output, while on more general distributions the program may gradually replicate the input at all nodes.

From this perspective, theoretical work is also needed on quantifying the costs required for computing certain queries. Example costs are the number of coordination steps [33], and the amount of replication [18].

Also interesting, is to investigate how increased local knowledge on nodes allows non-monotone computations to avoid coordination [55, 11].

Correctness.

The mentioned decidability results [13, 10] provide an indication that automatically deciding correctness of a distributed system might not be a useful strategy in practice. Indeed, we have to severely restrict expressivity to obtain decidability, and yet

[8]For completeness, we mention that this result depends on the assumption that only a finite number of messages arrive at each local timestamp of a node; a property also mentioned in Section 5.3.3.

the time complexity remains prohibitively high.

It might be more promising to achieve correctness through practical mechanisms and protocols [50, 16, 22], design guidelines [19], and insights about monotonicity such as the CALM conjecture [30, 7, 55, 12, 11].

Declarative semantics.

The motivation behind declarative networking is to simplify programming of cloud computing. Besides offering a convenient syntax, declarative networking should offer intuitive ways to think about the programmed functionality, also in asynchronous communication models. This could be done with a suitable declarative semantics, for which some initial explorations have been mentioned in Section 5.3.

We believe that declarative semantics for declarative networking deserves more attention. The difficulty is that a declarative semantics should at the same time hide technical operational details, whilst at the same time preserving an understanding of the distributed setting. In particular, further work seems needed to provide an intuitive semantics for distributed negation. This investigation could also encompass features like aggregation. But the intuitiveness of the semantics should not force execution engines to resort to heavy coordination. Perhaps this will only work for some restricted classes of programs.

Also, the stable models considered by previous work [5, 37, 6] are typically infinite, because they represent an infinite time domain in which the distributed computation unfolds. This is partly caused by asynchronous communication, where messages have arbitrary delays. In further work, it would be interesting to find finite representations of these stable models, again perhaps only for restricted classes of programs.

Regarding the CRON conjecture, further work is needed to understand the spectrum of causality: besides positive programs, perhaps richer classes of programs can tolerate some relaxations of causality as well. Also, it might be intriguing to link the CRON conjecture more concretely to crash recovery mechanisms, or other application scenarios giving rise to non-causality.

7. REFERENCES

[1] S. Abiteboul, E. Antoine, G. Miklau, J. Stoyanovich, and J. Testard. Rule-based application development using webdamlog. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 965–968. ACM, 2013.

[2] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for Web data management. In *Proceedings 30th ACM Symposium on Principles of Database Systems*, pages 293–304. ACM Press, 2011.

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

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

[5] P. Alvaro, T.J. Ameloot, J.M. Hellerstein, W.R. Marczak, and J. Van den Bussche. A declarative semantics for Dedalus. Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley, Nov 2011.

[6] P. Alvaro, T.J. Ameloot, J.M. Hellerstein, W.R. Marczak, and J. Van den Bussche. A declarative semantics for Dedalus. Hasselt University, Technical report, `http://hdl.handle.net/1942/14572`, 2013.

[7] P. Alvaro, N. Conway, J. Hellerstein, and W.R. Marczak. Consistency analysis in Bloom: A CALM and collected approach. In *Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260. www.cidrdb.org, 2011.

[8] P. Alvaro, N. Conway, J.M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. Technical Report UCB/EECS-2013-133, EECS Department, University of California, Berkeley, Jul 2013.

[9] P. Alvaro, W.R. Marczak, N. Conway, J.M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. In de Moor et al. [23], pages 262–281.

[10] T.J. Ameloot. Deciding correctness with fairness for simple transducer networks. In *Proceedings of the 17th International Conference on Database Theory*, 2014 (to appear).

[11] T.J. Ameloot, B. Ketsman, F. Neven, and D. Zinn. Weaker forms of monotonicity for declarative networking: a more fine-grained answer to the CALM-conjecture. In *Proceedings 33rd ACM Symposium on Principles of Database Systems*, 2014 (to appear).

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

[13] T.J. Ameloot and J. Van den Bussche. Deciding eventual consistency for a simple class of relational transducer networks. In *Proceedings of the 15th International Conference on Database Theory*, pages 86–98. ACM Press, 2012.

[14] T.J. Ameloot and J. Van den Bussche. Positive Dedalus programs tolerate non-causality. *Journal of Computer and System Sciences*, to appear.

[15] K.R. Apt and R.N. Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19-20, Supplement 1(0):9–71, 1994.

[16] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3), 2013.

[17] P. Barceló and R. Pichler, editors. *Datalog in Academia and Industry*, volume 7494 of *Lecture Notes in Computer Science*. Springer, 2012.

[18] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM Symposium on Principles of Database Systems*, pages 273–284. ACM Press, 2013.

[19] M. Cavage. There's just no getting around it: You're building a distributed system. *ACM Queue*, 11(4), 2013.

[20] J. Chomicki. Depth-bounded bottom-up evaluation of logic programs. *The Journal of Logic Programming*, 25(1):1–31, 1995.

[21] J. Chomicki and T. Imieliński. Finite representation of infinite query answers. *ACM Transactions on Database Systems*, 18(2):181–223, 1993.

[22] N. Conway, W.R. Marczak, P. Alvaro, J.M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1:1–1:14. ACM Press, 2012.

[23] O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors. *Datalog Reloaded: First International Workshop, Datalog 2010*, volume 6702 of *Lecture Notes in Computer Science*, 2011.

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

[25] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *Proceedings 12th International Conference on Database Theory*, 2009.

[26] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven Web applications. *Journal of Computer and System Sciences*, 73(3):442–474, 2007.

[27] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven Web services. In *Proceedings 25th ACM Symposium on Principles of Database Systems*, pages 90–99. ACM Press, 2006.

[28] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[29] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In M. Carro and R. Peña, editors, *Proceedings 12th International Symposium on Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 88–103, 2010.

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

[31] S.S. Huang, T.J. Green, and B.T. Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1213–1216. ACM, 2011.

[32] M. Interlandi, L. Tanca, and S. Bergamaschi. Datalog in time and space, synchronously. In L. Bravo and M. Lenzerini, editors, *AMW*, volume 1087 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[33] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *Proceedings of the 30th ACM symposium on Principles of Database Systems*, pages 223–234. ACM Press, 2011.

[34] G. Lausen, B. Ludäscher, and W. May. On active deductive databases: The statelog approach. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *Lecture Notes in Computer Science*, pages 69–106. Springer Berlin Heidelberg, 1998.

[35] J. Leite and L. Soares. Adding evolving abilities to a multi-agent system. In *Proceedings of the 7th International Conference on Computational Logic in Multi-agent Systems*, CLIMA VII'06, pages 246–265. Springer-Verlag, 2007.

[36] J.A. Leite, J.J. Alferes, and L.M. Pereira.

Minerva – a dynamic logic programming agent architecture. In *Revised Papers from the 8th International Workshop on Intelligent Agents VIII*, ATAL, pages 141–157. Springer-Verlag, 2002.

[37] J. Lobo, J. Ma, A. Russo, and F. Le. Declarative distributed computing. In E. Erdem, J. Lee, Y. Lierler, and D. Pearce, editors, *Correct Reasoning*, volume 7265 of *Lecture Notes in Computer Science*, pages 454–470. Springer, 2012.

[38] B.T. Loo, T. Condie, et al. Declarative networking: language, execution and optimization. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *SIGMOD Conference*, pages 97–108. ACM, 2006.

[39] B.T. Loo, T. Condie, et al. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009.

[40] B.T. Loo, H. Gill, C. Liu, Y. Mao, W.R. Marczak, M. Sherr, A. Wang, and W. Zhou. Recent advances in declarative networking. In C. Russo and N. Zhou, editors, *Practical Aspects of Declarative Languages*, volume 7149 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

[41] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J.M. Hellerstein. Graphlab: A new framework for parallel machine learning. In P. Grünwald and P. Spirtes, editors, *UAI*, pages 340–349. AUAI Press, 2010.

[42] L. Lu and J.G. Cleary. An operational semantics of starlog. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 294–310. Springer Berlin Heidelberg, 1999.

[43] J. Ma, F. Le, D. Wood, A. Russo, and J. Lobo. A declarative approach to distributed computing: Specification, execution and analysis. *Theory and Practice of Logic Programming*, 13:815–830, 2013.

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

[45] W.R. Marczak, P. Alvaro, N. Conway, J.M. Hellerstein, and D. Maier. Confluence analysis for distributed programs: A model-theoretic approach. In Barceló and Pichler [17], pages 135–147.

[46] J.A. Navarro and A. Rybalchenko. Operational semantics for declarative networking. In A. Gill and T. Swift, editors, *Proceedings 11th International Symposium on Practical Aspects of Declarative Languages*, volume 5419 of *Lecture Notes in Computer Science*, pages 76–90, 2009.

[47] V. Nigam, L. Jia, B.T. Loo, and A. Scedrov. Maintaining distributed logic programs incrementally. *Computer Languages, Systems & Structures*, 38(2):158–180, 2012.

[48] V. Nigam and J. Leite. A dynamic logic programming based system for agents with declarative goals. In *Proceedings of the 4th International Conference on Declarative Agent Languages and Technologies*, DALT, pages 174–190. Springer-Verlag, 2006.

[49] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 205–217. ACM Press, 1990.

[50] M. Shapiro, N.M. Preguiça, C. Baquero, and M. Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, 2011.

[51] M. Spielmann. Verification of relational transducers for electronic commerce. *Journal of Computer and System Sciences*, 66(1):40–65, 2003.

[52] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[53] C. Zaniolo, N. Arni, and K. Ong. Negation and aggregates in recursive rules: the ldl++ approach. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Deductive and Object-Oriented Databases*, volume 760 of *Lecture Notes in Computer Science*, pages 204–221. Springer Berlin Heidelberg, 1993.

[54] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 2010.

[55] D. Zinn, T.J. Green, and B. Ludäscher. Win-move is coordination-free (sometimes). In *Proceedings of the 15th International Conference on Database Theory*, pages 99–113. ACM Press, 2012.