TLP **18** (5–6): 806–845, 2018. © Cambridge University Press 2018
doi:10.1017/S1471068418000418

*Scaling-up reasoning and advanced analytics on BigData**

TYSON CONDIE, ARIYAM DAS, MATTEO INTERLANDI, ALEXANDER SHKAPSKY, MOHAN YANG and CARLO ZANIOLO

University of California, Los Angeles, CA, USA
(*e-mails:* tcondie@cs.ucla.edu, ariyam@cs.ucla.edu, minterlandi@cs.ucla.edu, shkapsky@cs.ucla.edu, yang@cs.ucla.edu, zaniolo@cs.ucla.edu)

submitted 15 May 2017; revised 12 July 2018; accepted 22 July 2018

Abstract

BigDatalog is an extension of Datalog that achieves performance and scalability on both Apache Spark and multicore systems to the point that its graph analytics outperform those written in GraphX. Looking back, we see how this realizes the ambitious goal pursued by deductive database researchers beginning 40 years ago: this is the goal of combining the rigor and power of logic in expressing queries and reasoning with the performance and scalability by which relational databases managed BigData. This goal led to Datalog which is based on Horn Clauses like Prolog but employs implementation techniques, such as semi-naïve fixpoint and magic sets, that extend the bottom-up computation model of relational systems, and thus obtain the performance and scalability that relational systems had achieved, as far back as the 80s, using data-parallelization on shared-nothing architectures. But this goal proved difficult to achieve because of major issues at (i) the language level and (ii) at the system level. The paper describes how (i) was addressed by simple rules under which the fixpoint semantics extends to programs using count, sum and extrema in recursion, and (ii) was tamed by parallel compilation techniques that achieve scalability on multicore systems and Apache Spark. This paper is under consideration for acceptance in Theory and Practice of Logic Programming.

KEYWORDS: Deductive databases, Datalog, BigData, parallel and distributed computing

1 Introduction

A growing body of research on scalable data analytics has brought a renaissance of interest in Datalog because of its ability to specify declaratively advanced data-intensive applications that execute efficiently over different systems and architectures, including massively parallel ones (Seo *et al.* 2013; Shkapsky *et al.* 2013; Yang and Zaniolo 2014; Aref *et al.* 2015; Wang *et al.* 2015; Yang *et al.* 2015; Shkapsky *et al.* 2016; Yang *et al.* 2017). The trends and developments that have led to this

* This work was supported in part by NSF under Grants IIS-1218471, IIS-1302698 and CNS-1351047, and in part by NIH BigData to Knowledge (BD2K) under Grant U54EB020404.

renaissance can be better appreciated if we contrast them with those that motivated the early research on Datalog back in the 80s. The most obvious difference is the great importance and pervasiveness of BigData that, by enabling intelligent decision making and solving complex problems, is delivering major benefits to societies and economies. This is remarkably different from the early work on Datalog in the 80s, which was motivated by interest in expert system applications that then proved to be only of transient significance. The main objective of this paper is to present the significant technological advances that have made possible for Datalog to exploit the opportunities created by BigData applications. One is the newly found ability to support a larger set of applications by extending the declarative framework of Horn clauses to include aggregates in recursive rules. The other is the ability of scaling up Datalog applications on BigData by exploiting parallel systems featuring multicore and distributed architectures. We will next introduce and discuss the first topic, by summarizing the recent findings presented in Zaniolo *et al.* (2017) and then extending them with new examples of graph algorithms, and knowledge discovery and data mining applications. The second topic is briefly discussed in this section; then, it is revisited in Section 5 and fully discussed in Sections 6 and 7 on the basis of results and techniques from Shkapsky *et al.* (2016) and Yang *et al.* (2017).

A common trend in the new generation of Datalog systems is the usage of aggregates in recursion, since they enable the concise expression and efficient support of much more powerful algorithms than those expressible by programs that are stratified w.r.t. negation and aggregates (Seo *et al.* 2013; Shkapsky *et al.* 2013; Wang *et al.* 2015; Shkapsky *et al.* 2016). As discussed in more detail in the related work section, extending the declarative semantics of Datalog to allow aggregates in recursion represents a difficult problem that had seen much action in the early days of Datalog (Kemp and Stuckey 1991; Greco *et al.* 1992; Ross and Sagiv 1992). Those approaches sought to achieve both (i) a formal declarative semantics for deterministic queries using the basic SQL aggregates, min, max, count and sum, in recursion and (ii) their efficient implementation by extending techniques of the early Datalog systems (Morris *et al.* 1986; Chimenti *et al.* 1987; Ramakrishnan *et al.* 1992; Vaghani *et al.* 1994; Arni *et al.* 2003). Unfortunately, as discussed in the related work section, some of those approaches had limited generality since they did not deal with all four basic aggregates, while the proposal presented in Ross and Sagiv (1992) that was covering all four basic aggregates using different lattices for different aggregates faced other limitations, including those pointed out by Van Gelder (1993) that are discussed in Section 8. These works were followed by more recent approaches that addressed the problem of using more powerful semantics, such as answer-set semantics, that require higher levels of computational complexity and thus are a better for higher-complexity problems than for the very efficient algorithms needed on BigData (Simons *et al.* 2002; Pelov *et al.* 2007; Son and Pontelli 2007; Swift and Warren 2010; Faber *et al.* 2011; Gelfond and Zhang 2014).

The recent explosion of work on BigData has also produced a revival of interest in Datalog as a parallelizable language for expressing and supporting efficiently BigData Analytics (Seo *et al.* 2013; Shkapsky *et al.* 2013; Wang *et al.* 2015). As

described from Section 5 onward, the projects discussed in those papers have demonstrated the ability of Datalog to provide scalable support for BigData applications on both multicore and distributed systems. Most of the algorithms discussed in those papers are graph algorithms or other algorithms that use aggregates in recursion, whereby a full convergence of formal declarative semantics and amenability to efficient implementation becomes a critical objective. By supporting graph applications written in Datalog and compiled onto Apache Spark with better performance than the same applications written in GraphX (a Spark framework optimized for graph algorithms) and Scala (Spark's native language), our BigDatalog system (Shkapsky *et al.* 2016) proved that we have achieved this very difficult objective. Along with the post-MapReduce advances demonstrated by Apache Spark, this success was made possible by the theoretical developments presented in Zaniolo *et al.* (2017), where the concept of *premappability* (*PreM*) was introduced for constraints using a unifying semantics that makes possible the use of the aggregates min, max, count and sum in recursive programs. Indeed, *PreM* of constraints provides a simple criterion that (i) the system optimizer can utilize to push constraints into recursion, and (ii) the user can utilize to write programs using aggregates in recursion, with the guarantee that they have indeed a formal fixpoint semantics. Along with its formal fixpoint semantics, this approach also extends the applicability of traditional Datalog optimization techniques, to programs that use aggregates in rules defining recursive predicates.

The rest of this paper is organized as follows. In the next section, we introduce the problem of supporting aggregates in recursion; then, in Section 3, we present how such Datalog extension can be used in practice to implement efficient graph applications. We thus introduce in Section 4 even more advanced knowledge discovery and data mining analytics such as classification and regression. Sections 5–7 introduce our BigDatalog and BigDatalog-MC systems that support scalable and efficient analytics through distributed and multicore architectures, respectively. Related work and conclusion presented in Sections 8 and 9 bring the paper to a closing.

2 Datalog extensions: min and max in recursive rules

In this section, we first introduce some basics about Datalog before explaining its recent extensions. A Datalog program is formally represented as a finite set of rules. A Datalog rule, in turn, can be represented as $h \leftarrow b_1, \ldots, b_n$, where h denotes the head of the rule and $b_1, \ldots, b_n$ represents the corresponding body. Technically, h and each b_i are literals assuming the form $p_i(t_1, \ldots, t_j)$, where p_i is a predicate and each t_i can either be a constant or a variable. A rule with an empty body is called a *fact*. The comma separating literals in a body of the rule represents logical conjunction (AND). Throughout the paper, we follow the convention that predicate and function names begin with lower case letters, and variable names begin with upper case letters.

A most significant advance in terms of language and expressive power offered by our systems (Shkapsky *et al.* 2016; Yang *et al.* 2017) is that they provide a formal

semantics and efficient implementation for recursive programs that use min, max, count and sum in recursion. We present here an informal summary of these advances for which Zaniolo *et al.* (2017) provides a formal in-depth coverage.

Consider for instance Example 1, where the goal `is_min((X, Z), (Dxz))` in r_3 specifies that we want the min values of `Dxz` for each unique pair of values `(X, Z)` in `dpath` defined by rules r_1 and r_2.

Example 1 (*Computing distances between node pairs, and finding their min*)

$$r_1 : \texttt{dpath}(\texttt{X}, \texttt{Z}, \texttt{Dxz}) \texttt{ <- } \texttt{darc}(\texttt{X}, \texttt{Z}, \texttt{Dxz}).$$
$$r_2 : \texttt{dpath}(\texttt{X}, \texttt{Z}, \texttt{Dxz}) \texttt{ <- } \texttt{dpath}(\texttt{X}, \texttt{Y}, \texttt{Dxy}), \texttt{darc}(\texttt{Y}, \texttt{Z}, \texttt{Dyz}), \texttt{Dxz} = \texttt{Dxy} + \texttt{Dyz}.$$
$$r_3 : \texttt{spath}(\texttt{X}, \texttt{Z}, \texttt{Dxz}) \texttt{ <- } \texttt{dpath}(\texttt{X}, \texttt{Z}, \texttt{Dxz}), \texttt{is\_min}((\texttt{X}, \texttt{Z}), (\texttt{Dxz})).$$

Thus, the special notation `is_min((X, Z), (Dxz))` tells the compiler that `spath(X, Z, Dxz)` is a special predicate supported by a specialized implementation (the query and optimization techniques will be discussed in the following sections). Similar observations also hold for `is_max`. However, the formal semantics of rules with extrema constructs is defined using standard (closed-world) negation, whereby the semantics of r_3 is defined by the following two rules[1].

$$\texttt{spath}(\texttt{X}, \texttt{Z}, \texttt{Dxz}) \texttt{ <- } \texttt{dpath}(\texttt{X}, \texttt{Z}, \texttt{Dxz}), \neg\texttt{lesser}(\texttt{X}, \texttt{Z}, \texttt{Dxz}).$$
$$\texttt{lesser}(\texttt{X}, \texttt{Z}, \texttt{Dxz}) \texttt{ <- } \texttt{dpath}(\texttt{X}, \texttt{Z}, \texttt{Dxz}), \texttt{dpath}(\texttt{X}, \texttt{Z}, \texttt{D1}), \texttt{D1} < \texttt{Dxz}.$$

Expressing `is_min` via negation also reveals the non-monotonic nature of extrema constrains, whereby this program will be treated as a stratified program, with a "perfect model" semantics, realized by an iterated-fixpoint computation (Przymusinski 1988). In this computation, `dpath` is assigned to a stratum lower than `spath` and thus the computation of `dpath` must complete before the computation of `spath` via `is_min` in r_3 can begin. This stratified computation can be very inefficient or even non-terminating when the original graph of Example 1 contains cycles. Thus, much research work was spent on solving this problem, before the simple solution described next emerged, and was used in BigDatalog to support graph algorithms with superior performance (Shkapsky *et al.* 2016). This solution consists in taking the `is_min` constraint on `dpath` in r_3 and moving it to the rules r_1 and r_2 defining `dpath`, producing the rules in Example 2. This rewriting will be called a *transfer* of constraints[2].

[1] This rewriting assumes that there is only one `is_min` in our program. In the presence of multiple occurrences, we will need to add a subscript to keep them distinct.

[2] In the example at hand, we have conveniently used the same names for corresponding variables in all our rules. In general, however, the transfer also involves a renaming for the variable(s) used in specifying the constraint.

Example 2 (*Shortest distances between node pairs*)

r'_1 : dpath(X, Z, Dxz) <- darc(X, Z, Dxz), is_min((X, Z), (Dxz)).
r'_2 : dpath(X, Z, Dxz) <- dpath(X, Y, Dxy), darc(Y, Z, Dyz),
Dxz = Dxy + Dyz, is_min((X, Z), (Dxz)).
r'_3 : spath(X, Z, Dxz) <- dpath(X, Z, Dxz).

While, at the syntactic level, this transfer of constraint is quite simple, at the semantic level, it raises the following two critical questions: (i) does the program in Example 2 have a formal semantics, notwithstanding the fact that it uses non-monotonic constructs in recursion, and (ii) is it equivalent to the original program, insofar as it produces the same answers for spath? A positive formal answer to both questions was provided in Zaniolo *et al.* (2017) using the notion of *premappability* (*PreM*), which is summarized next.

Premappability (PreM) for constraints. Let T denote the immediate consequence operator (ICO) for the rules defining a recursive predicate[3]. Since our rules are positive, the mapping defined by T has a least-fixpoint in the lattice of set containment. Moreover, the property that such least-fixpoint is equivalent to the fixpoint iteration $T^{\uparrow\omega}(\emptyset)$ allows us to turn this declarative semantics into a concrete one. Now, let γ be a constraint, such as an extrema constraint like min. We have the following important definition:

Definition 1
The constraint γ is said to be *PreM* to T when, for every interpretation I, we have that: $\gamma(T(I)) = \gamma(T(\gamma(I)))$.

For convenience of notation, we will also denote by T_γ the composition of the function T with the function γ, i.e., $T_\gamma(I) = \gamma(T(I))$, which will be called the *constrained ICO* for T and the rules having T as their ICO. Then, *PreM* holds whenever $T_\gamma(I) = T_\gamma(\gamma(I))$. We will next focus on cases of practical interest where the transfer of constraints under *PreM* produces optimized programs that are safe and terminating (even when the original programs do not terminate). Additionally, we prove that the transformation is indeed equivalence-preserving. Thus, we focus on situations where $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$, i.e., the fixpoint iteration converges after a finite number of steps n. The rules defining a recursive predicate p are those having as head p or predicates that are mutually recursive with p. Then, the following theorem was proven in Zaniolo *et al.* (2017):

Theorem 1
In a Datalog program, let T be the ICO for the positive rules defining a recursive predicate. If the constraint γ is *PreM* to T, and a fixpoint exists such that $T^{\uparrow n+1}(\emptyset) = T^{\uparrow n}(\emptyset)$ for some integer n, then $\gamma(T^{\uparrow\omega}(\emptyset)) = T_\gamma^{\uparrow n}(\emptyset)$.

In Zaniolo *et al.* (2017), it was also shown that the fixpoint so derived is a minimal fixpoint for the program produced by the transfer of constraints. Thus, if a constraint

[3] The case of multiple mutually recursive predicates will be discussed later.

is *PreM* to the given recursive rules, its transfer produces an optimized program having a declarative semantics defined by the minimal fixpoint of its constrained ICO (T_γ) and operational semantics supported by a terminating fixpoint iteration, with all the theoretical and computational properties that follow from such semantics. For instance, *PreM* for extrema constraints holds for Example 1, and since directed arcs in our graph have non-negative lengths, we conclude that its optimized version in Example 2 terminates even if the original graph has cycles.

For most applications of practical interest, *PreM* is simple for users to program with, and for the system to support[4]. For instance, to realize that *PreM* of min and max holds for the rules for our Example 1, the programmer will test *PreM* by asking how the mapping established by rules $\mathtt{r}'_1$ and $\mathtt{r}'_2$ in Example 2 changes if, in addition to the post-constraint `is_min` that applies to the cost arguments of the head of rules $\mathtt{r}'_1$ and $\mathtt{r}'_2$, we add the goal `is_min` in the body of our two rules to pre-constrain the values of the cost argument in every `dpath` goal. Of course, *PreM* is trivially satisfied in $\mathtt{r}'_1$ since this is an exit rule with no `dpath` goal, whereby the rule and its associate mapping remain unchanged. In rule $\mathtt{r}'_2$, the application of the pre-constraint $\mathtt{is\_min}((\mathtt{X},\mathtt{Y}),\mathtt{Dxy})$ to the values generated by $\mathtt{dpath}(\mathtt{X},\mathtt{Y},\mathtt{Dxy})$ does not change the final values returned by this rule because of the arithmetic properties of its interpreted goals $\mathtt{Dxz} = \mathtt{Dxy} + \mathtt{Dyz}$; in fact, these assure that every $\overline{\mathtt{Dxy}} > \mathtt{Dxy}$ can be eliminated since the value $\overline{\mathtt{Dxz}} = \overline{\mathtt{Dxy}} + \mathtt{Dyz}$ it produces is higher than `Dxz` and will thus be eliminated by the $\mathtt{is\_min}((\mathtt{X},\mathtt{Z}),\mathtt{Dxz})$ post-constraint.

This line of reasoning is simple enough for the programmer to understand and for the system to verify. More general conditions for *PreM* are given in Zaniolo *et al.* (2017) using the notions of inflation-preserving and deflation-preserving rules. There, we also discuss the premappability of the lower-bound and the upper-bound constraints, which are often used in conjunction with extrema, and interact with them to determine *PreM* and the termination of the resulting program. For instance, to find the maximum distance between nodes in a graph that is free of directed cycles, the programmer will simply replace `is_min` with `is_max` in Examples 1 and 2 with the assurance that the second program so obtained is the optimized equivalent of the first since (i) premappability holds, and (ii) its computation terminates in a finite number of steps[5]. However, say that the programmer wants to add to the recursive rule of this second program the condition $\mathtt{Dxz} < \mathtt{Upperbound}$ either because (i) only results that satisfy this inequality are of interest, or (ii)

[4] In fact, premappability is a very general property that has been widely used in advanced analytics under different names and environments. For instance, the antimonotonic property of frequent item sets represents just a particular form of premappability that will be discussed in Section 4. Also, with `OP` denoting sum or min or max, we have that

$$\mathtt{OP}\Big(\bigcup_{1\leqslant j\leqslant K} S_j\Big) = \mathtt{OP}\Big(\bigcup_{1\leqslant j\leqslant K} \mathtt{OP}(S_j)\Big)$$

Thus, `OP` is premappable w.r.t. union; this is the pre-aggregation property that is commonly used in distributed processing since it delivers major optimizations (Yu *et al.* 2009).

[5] Besides representing a practical requirement in applications, termination is also required from a theoretical viewpoint since, for programs such as that of Example 2, a stable model exists if and only if it has a termination (A. Das and M. Interlandi, personal communication).

this precautionary step is needed to guarantee termination when fortuitous cycles are created by accidental insertions of wrong data[6]. However, if the condition $\texttt{Dxz} < \texttt{Upperbound}$ is added as a goal to recursive rule of our program, its *PreM* property is compromised. To solve this problem the Datalog programmer should instead replace $\texttt{Dyz=Dxz+Dxy}$ with the following condition:

$$\texttt{if(Dxy+Dyz} > \texttt{Upperbound then Dxz=Upperbound else Dxz=Dxy+Dyz)}$$

This condition can be expressed as such in our systems, or can be re-expressed using a pair of positive rules in other Datalog systems. This formulation ensures termination while preserving *PreM* for max constraints. Symmetrically, the addition of lower-bound constraints in our Example 2 must be performed in a similar way to avoid compromising *PreM*.

Our experience suggests that using the insights gained from these simple examples, a programmer can master the use of *PreM* constraints to express significant algorithms in Datalog, with assurance they will deliver performance and scalability.

In the next example, we present a non-linear version of Example 1, where we use the head notation for aggregates that is supported in our system.

Example 3 (*Shortest distances between node pairs*)

$(\texttt{r}_4)$ $\texttt{dpath(X, Z, min}\langle\texttt{Dxz}\rangle\texttt{)} \texttt{ <- darc(X, Z, Dxz), Dxz} > 0.$

$(\texttt{r}_5)$ $\texttt{dpath(X, Z, min}\langle\texttt{Dxz}\rangle\texttt{)} \texttt{ <- dpath(X, Y, Dxy), dpath(Y, Z, Dyz), Dxz} = \texttt{Dxy+Dyz.}$

The special head notation, is in fact a short hand for adding final goal $\texttt{is\_min((X, Z), (Dxz))}$ that still defines the formal semantics of our rules. Therefore, *PreM* for $\texttt{r}_5$ is determined by adding the pre-constraints $\texttt{is\_min((X, Y), (Dxy))}$ and $\texttt{is\_min((Y, Z), (Dxy))}$, respectively, after the first and the second goal and asking if these changes affect the final values that survive the post-constraint in the head of the rule. Here again, the values $\overline{\texttt{Dxy}} > \texttt{Dxy}$ and values $\overline{\texttt{Dyz}} > \texttt{Dyz}$ can be eliminated without changing the head results once the post-constraint is applied.

2.1 From monotonic count to regular COUNT and SUM

At the core of the approach proposed in Mazuran *et al.* (2013b) there is the observation that the cumulative version of standard count is monotonic in the lattice of set containment. Thus, the authors introduced `mcount` as the aggregate function that returns all natural numbers up to the cardinality of the set. The use of `mcount` in actual applications is illustrated by the following example that uses the motonic count `mcount` in the head of rules to express an application similar to the one proposed in Ross and Sagiv (1992).

[6] For example, Bill of Materials databases store, for each part in the assembly, its subparts with their quantities. Bill of Materials databases define acyclic directed graphs; but the risk of some bad data can never be ruled out in such databases containing millions of records.

Example 4 (*Join the party once you see that three of your friends have joined*)
The organizer of the party will attend, while other people will attend if the number of their friends attending is greater or equal to 3, i.e., $\texttt{Nfx} \geqslant 3$.

$$\begin{aligned}
&\mathtt{r}_6 : \texttt{attend(X)} \texttt{ <- } \texttt{organizer(X)}.\\
&\mathtt{r}_7 : \texttt{attend(X)} \texttt{ <- } \texttt{cntfriends(X, Nfx)}, \texttt{Nfx} \geqslant 3.\\
&\mathtt{r}_8 : \texttt{cntfriends(Y, mcount}\langle\texttt{X}\rangle) \texttt{ <- } \texttt{attend(X)}, \texttt{friend(Y, X)}.\\
&\mathtt{r}_9 : \texttt{finalcnt(Y, max}\langle\texttt{N}\rangle) \texttt{ <- } \texttt{cntfriends(Y, N)}.
\end{aligned}$$

As described in Mazuran *et al.* (2013b), the formal semantics of `mcount` can be reduced to the formal semantics of Horn Clauses. Thus, `mcount` is a monotonic aggregate function and as such is fully compatible with standard semantics of Datalog and its optimization techniques, including the transfer of extrema, discussed in the previous section. In terms of operational semantics, however, `mcount` will enumerate new friends one at the time and could be somewhat slow. An obvious alternative consists in premapping the max value to `mcount` since the combination of `mcount` and `max` defines the traditional count. Then, in the fixpoint computation, the new count value will be upgraded to the new max, rather than the succession of $+1$ upgrades computed by `mcount`. Thus, the rules $\mathtt{r}_8, \mathtt{r}_9$ can be substituted with $\mathtt{r}'_8, \mathtt{r}'_9$, respectively, as follows:

$$\begin{aligned}
&\mathtt{r}'_8 : \texttt{cntfriends(Y, count}\langle\texttt{X}\rangle) \texttt{ <- } \texttt{attend(X)}, \texttt{friend(Y, X)}.\\
&\mathtt{r}'_9 : \texttt{finalcnt(Y, N)} \texttt{ <- } \texttt{cntfriends(Y, N)}.
\end{aligned}$$

The question of whether `max` is *PreM* to our rules can be formulated by assuming that we apply a vector of constraints one for each mutually recursive predicate. Thus, in Example 4, we will apply the max constraint to `cntfriends` and a null constraint, that we will call `nofilter`, to `attend`. Now, the addition of `nofilter(X)` does not change the mapping defined by $\mathtt{r}_8$, and the addition of `is_max(X, Nfx)` does not change the mapping defined by $\mathtt{r}_7$ since the condition $\texttt{Nfx} \geqslant 3$ is satisfied for some `Nfx` value iff it is satisfied by the max of these values. Thus, *PreM* is satisfied and `mcount` in $\mathtt{r}_8$ can be replaced by the regular `count`.

From monotonic SUM to SUM. The notion of monotonic sum, i.e., `msum`, for positive numbers introduced in Mazuran *et al.* (2013b) used the fact that its semantics can be easily reduced to that of `mcount`, as illustrated by the example below that computes the total number of each part available in a city by adding up the quantities held in each store of that city:

$$\texttt{pCnt\_InCity(Pno, City, sum}\langle\texttt{Qty, Store}\rangle) \texttt{ <- } \texttt{pqs(Pno, Store, Qty)}, \texttt{cs(Store, City)}.$$

Here, the sum is computed by adding up the `Qty` values, but the presence of `Store` makes sure that all the repeated occurrences of the same `Qty` are considered in the addition, rather than being ignored as a set semantics would imply. The results returned by this rule are the same as those returned by the following rule where `posint` simply enumerates the positive integers up to `Qty`:

$$\begin{aligned}
&\texttt{partCnt_InCity(Pno, City, count}\langle\texttt{Eq, Store}\rangle) \texttt{ <-}\\
&\qquad\qquad \texttt{pqs(Pno, Store, Qty)}, \texttt{cs(Store, City)}, \texttt{posint(Qty, Eq)}.
\end{aligned}$$

Then, consider the following example where we want to count the distinct paths connecting any pair of node in the graph:

```
cpath(X, X, 1) <- arc(X, _).
cpath(X, Z, sum⟨Cxy, Y⟩) <- cpath(X, Y, Cxy), arc(Y, Z).
```

Then, the semantics of our program is defined by its equivalent rewriting

Example 5 (*Sum of positive numbers expressed via count*)

```
cpath(X, Y, 1) <- edge(X, Y).
cpath(X, Z, count⟨Y, Ixy⟩) <- cpath(X, Y, Cxy), edge(Y, Z), posint(Cxy, Ixy).
```

Thus, whenever a sum aggregate is used, the programmer and the compiler will determine its correctness, by (i) replacing sum with msum, (ii) replacing msum by mcount via the `posint` expansion and (iii) checking that the max aggregate is *PreM* in the program so rewritten. Of course, once this check succeeds, the actual implementation uses the sum aggregate directly, rather than its equivalent, due to the inefficient expansion of the count aggregator. While, in this example, we have used positive integers for cost arguments, the sum of positive floating point numbers can also be handled in the same fashion (Mazuran *et al.* 2013b).

3 In-database graph applications

The use of aggregates in recursion has allowed to express efficiently a wide spectrum of applications that were very difficult to express and support in traditional Datalog. Several graph and mixed graph-relation applications were described in Shkapsky *et al.* (2016) and Yang (2017). Other applications[7] include, the Viterbi algorithm for hidden Markov models, connected components by label propagation, temporal coalescing of closed periods, the people you know, the multilevel marketing network bonus calculation and several bill of materials queries such as parts, costs and days required in an assembly. Two new graph applications that we have recently developed are given next, and advanced analytics and data mining applications are discussed in the next section.

Example 6 (*Diameter estimation*)
Many graph applications, particularly those appearing in the social networks setting, need to make an estimation about the diameter of its underlying network in order to complete several critical graph mining tasks like tracking evolving graphs over time (Kang *et al.* 2011). The traditional definition of the diameter as the farthest distance between two connected nodes is often susceptible to outliers. Hence, we compute the *effective diameter* (Kang *et al.* 2011), which is defined as follows: the effective diameter d of a graph G is formally defined as the minimum number of hops in which 90% of all connected pairs of nodes can reach each other. This measure is tightly related to closeness centrality and in fact is widely adopted in

[7] Programs available at `http://wis.cs.ucla.edu/deals/`

many network mining tasks (Cardoso *et al.* 2009). The following Datalog program shows how effective diameter can be estimated using aggregates in recursion.

$$r_{6.1} : \texttt{hops(X, Y, H)} \texttt{ <- } \texttt{arc(X, Y), H = 1.}$$

$$r_{6.2} : \texttt{hops(X, Y, min}\langle\texttt{C}\rangle\texttt{)} \texttt{ <- } \texttt{hops(X, Z, C1), hops(Z, Y, C2), C = C1 + C2.}$$

$$r_{6.3} : \texttt{minhops(X, Y, C)} \texttt{ <- } \texttt{hops(X, Y, C).}$$

$$r_{6.4} : \texttt{totalpairs(count}\langle\texttt{X}\rangle\texttt{)} \texttt{ <- } \texttt{minhops(X, \_, \_).}$$

$$r_{6.5} : \texttt{cumulhops(C, count}\langle\texttt{(X, Y)}\rangle\texttt{)} \texttt{ <- } \texttt{minhops(X, Y, C).}$$

$$r_{6.6} : \texttt{cumulhops(H2, sum}\langle\texttt{(H1, C)}\rangle\texttt{)} \texttt{ <- } \texttt{cumulhops(H1, C1), cumulhops(H2, C2),}$$
$$\texttt{H1 < H2, C = C1 + C2.}$$

$$r_{6.7} : \texttt{effdiameter(min}\langle\texttt{H}\rangle\texttt{)} \texttt{ <- } \texttt{cumulhops(H, C), totalpairs(N), C/N} \geqslant 0.9.$$

Rules $r_{6.1}$–$r_{6.3}$ find the minimum number of hops for each connected pair of vertices whereas rules $r_{6.5}$–$r_{6.6}$ compute the cumulative distribution of hops recursively using the fact that any pair of connected vertices covered within $H1$ hops is also covered in $H2$ hops ($H1 < H2$). The final rule $r_{6.7}$ extracts the effective diameter as per its definition (Kang *et al.* 2011).

Example 7 (*k-cores determination*)
A k-core of a graph G is a maximal connected subgraph of G in which all vertices have degree of at least k. k-core computation (Matula and Beck 1983) is critical in many graph applications to understand the clustering structure of the networks and is frequently used in bioinformatics and in many network visualization tools (Shin *et al.* 2016). The following Datalog program computes all the k-cores of a graph for an input k. Using aggregates in recursion in the following computation we determine all the connected components of the corresponding subgraph with degree k or more.

$$r_{7.1} : \texttt{degree(X, count}\langle\texttt{Y}\rangle\texttt{)} \texttt{ <- } \texttt{arc(X, Y).}$$

$$r_{7.2} : \texttt{validArc(X, Y)} \texttt{ <- } \texttt{arc(X, Y), degree(X, D1), D1} \geqslant \texttt{k,}$$
$$\texttt{degree(Y, D2), D2} \geqslant \texttt{k.}$$

$$r_{7.3} : \texttt{connComp(A, A)} \texttt{ <- } \texttt{validArc(A, \_).}$$

$$r_{7.4} : \texttt{connComp(C, min}\langle\texttt{B}\rangle\texttt{)} \texttt{ <- } \texttt{connComp(A, B), validArc(A, C).}$$

$$r_{7.5} : \texttt{kCores(A, B)} \texttt{ <- } \texttt{connComp(A, B).}$$

Example 7 determines k-cores by determining all the connected components ($r_{7.3}$, $r_{7.4}$), considering only vertices with degree k or more ($r_{7.1}$, $r_{7.2}$). The lowest vertex ID is selected as the connected component ID among the k-cores.

4 Advanced analytics

The application area of ever-growing importance, advanced analytics, encompass applications using standard online analytical processing (OLAP) to complex data mining and machine learning queries like frequent itemset mining (Agrawal *et al.* 1994), building classification models, etc. This new generation of advanced analytics

Table 1. *Training examples for the PlayTennis table*

ID	Outlook (1)	Temperature (2)	Humidity (3)	Wind (4)	Play tennis (5)
1	Overcast	Cool	Normal	Strong	Yes
2	Overcast	Hot	High	Weak	Yes
3	Overcast	Hot	Normal	Weak	Yes
4	Overcast	Mild	High	Strong	Yes
5	Rain	Mild	High	Weak	Yes
6	Rain	Cool	Normal	Weak	Yes
7	Rain	Cool	Normal	Strong	No
8	Rain	Mild	High	Strong	No
9	Rain	Mild	Normal	Weak	Yes
10	Sunny	Hot	High	Weak	No
…	…	…	…	…	…

is extremely useful in extracting meaningful and rich insights from data (Agrawal *et al.* 1994). However, these advanced analytics have created major challenges to database researchers (Agrawal *et al.* 1994) and the Datalog community (Giannotti and Manco 2002; Arni *et al.* 2003; Giannotti *et al.* 2004; Borkar *et al.* 2012). The major success that BigDatalog has achieved on graph algorithms suggests that we should revisit this hard problem and look beyond the initial applications discussed in Tsur (1991) by leveraging on the new opportunities created by the use of aggregates in recursion. We next describe briefly the approach we have taken and the results obtained so far.

Verticalized representation. First, we need to specify algorithms that can support advanced analytics on tables with arbitrary number of columns. A simple way to achieve this genericity is to use verticalized representations for tables. For instance, consider the excerpt from the well-known *PlayTennis* example from Mitchell (1997), shown in Table 1. The corresponding verticalized view is presented in Table 2, where each row contains the original tuple ID, a column number and the value of the corresponding column, respectively. The verticalization of a table with n columns (excluding the ID column) can be easily expressed by n rules; however, a special "@" construct is provided in our language to expedite this task. The use of this special construct is demonstrated by the rule below, which converts Table 1 into the verticalized view of Table 2.

```
vtrain(ID, Col, Val) <- train(ID, Val@Col).
```

Given a vertical representation, a simple data mining algorithm such as naive Bayesian classifiers (Lewis 1998) can be expressed by simple non-recursive rules[8]. However, a more advanced compact representation is needed to support complex tasks efficiently, as outlined next.

[8] http://wis.cs.ucla.edu/deals/tutorial/nbc.php

Table 2. *Vertical view of the tuples in Table 1*

ID	Col	Val
1	1	Overcast
1	2	Cool
1	3	Normal
1	4	Strong
1	5	Yes
2	1	Overcast
2	2	Hot
2	3	High
2	4	Weak
2	5	Yes
…	…	…

Rollup prefix table. To support efficiently more complex algorithms, such as frequent itemset mining (Agrawal *et al.* 1994) and decision tree construction (Quinlan 1986), we use an intuitive prefix-tree like representation that is basically a compact representation of the SQL-2003 COUNT ROLLUP[9] aggregate. For instance, the count rollup on Table 1 yields the output of Table 3, where we limit the output to the first 14 lines.

Interestingly, the output of ROLLUP contains many redundant `null` values and only the items in the main diagonal hold new information (highlighted in red). In fact, the items to left of the diagonal are repeating the previous values (i.e., sharing the prefix), whereas those to right are `nulls`. With this observation, we can compact Table 3 to a more logically concise representation shown in Table 4, where the first four columns contain the same information as an item in the main diagonal does, whereas the last column (PID) specifies the ID of the parent tuple from where we can find the value of the previous column. We refer to this condensed representation as a *prefix table* since it is in fact a table representation of the well-known prefix tree data structure. In this particular case, we have a *rollup prefix table* for count, and similar representations can be used for other aggregates.

An easier way to understand and visualize Table 4 is through the logically equivalent and more user intuitive representation, *compact rollups*, shown in Table 5 (equivalent values in Tables 3–5 are marked in red). In this representation, each item e that is not under the ID column and is not empty, represents a tuple in the rollup prefix table, where the values for ID, Col, Val, count columns are the tuple ID of e, e's column number, e's value and the number associated with e's value, respectively. Thus, Table 4 captures in a verticalized form the information that in a horizontal form is displayed by Table 5. In turn, this is a significantly compressed version of Table 3.

These rollups are simple to generate from our verticalized representation and they provide a good basis for programming other analytics (Das and Zaniolo

[9] https://technet.microsoft.com/en-us/library/bb522495(v=sql.105).aspx

Table 3. *The* SQL-2003 COUNT ROLLUP *on Table 1*

RID	Outlook (1)	Temp. (2)	Humidity (3)	Wind (4)	Play (5)	Count
1	Null	Null	Null	Null	Null	14
2	Overcast	Null	Null	Null	Null	4
3	Overcast	Cool	Null	Null	Null	1
4	Overcast	Cool	Normal	Null	Null	1
5	Overcast	Cool	Normal	Strong	Null	1
6	Overcast	Cool	Normal	Strong	Yes	1
7	Overcast	Hot	Null	Null	Null	2
8	Overcast	Hot	High	Null	Null	1
9	Overcast	Hot	High	Weak	Null	1
10	Overcast	Hot	High	Weak	Yes	1
11	Overcast	Hot	Normal	Null	Null	1
12	Overcast	Hot	Normal	Weak	Null	1
13	Overcast	Hot	Normal	Weak	Yes	1
14	Overcast	Mild	Null	Null	Null	1
…	…	…	…	…	…	…

Table 4. *A* rollup prefix table

ID	Col	Val	Count	PID
1	1	Null	14	1
2	1	Overcast	4	1
3	2	Cool	1	2
4	3	Normal	1	3
5	4	Strong	1	4
6	5	Yes	1	5
7	2	Hot	2	2
8	3	High	1	7
9	4	Weak	1	8
10	5	Yes	1	9
11	3	Normal	1	7
12	4	Weak	1	11
13	5	Yes	1	12
14	2	Mild	1	2
…	…	…	…	…

Table 5. *A compact rollup for the example in Table 1*

Outlook	C1	Temperature	C2	Humidity	C3	Wind	C4	Play	C5
Overcast	4	Cool	1	Normal	1	Strong	1	Yes	1
		Hot	2	High	1	Weak	1	Yes	1
				Normal	1	Weak	1	Yes	1
		Mild	1	High	1	Strong	1	Yes	1

2016; Yang 2017). We illustrate the construction of the rollup prefix table from the corresponding verticalized representation, using the rules described in the next example, which exploits aggregates in recursion.

Example 8 (*From a verticalized view* `vtrain` *to a rollup prefix table*)
Given two rows `T1` and `T2`, we say that the row `T1` can *represent* the row `T2` (or `T1` can represent `T2` for short) for the first `C` columns if both rows are identical in the first `C` columns (i.e., their prefixes are the same). `repr` is a recursive relation that represents `vtrain` in a different format, where each tuple $(\mathtt{T}, \mathtt{C}, \mathtt{V})$ in `vtrain` is augmented with one more column `T1` indicating that `T1` can represent `T` in first $\mathtt{C} - 1$ columns, i.e., the parent ID of the current row is `T1`. Then, a prefix table `rupt` is constructed (without the node count) on top of `repr` in $\mathtt{r}_1$, where among all the rows with the same parent ID `Ta`, and the same value `V` in column `C`, the one with the minimal ID `T` is selected as a *representative* by the aggregate min.

$$r_{8.1} : \mathtt{repr}(\mathtt{T1}, \mathtt{C}, \mathtt{V}, \mathtt{T}) \leftarrow \mathtt{vtrain}(\mathtt{T}, \mathtt{C}, \mathtt{V}), \mathtt{C} = 1, \mathtt{T1} = 1.$$

$$r_{8.2} : \mathtt{rupt}(\mathtt{min}\langle \mathtt{T} \rangle, \mathtt{C}, \mathtt{V}, \mathtt{Ta}) \leftarrow \mathtt{repr}(\mathtt{Ta}, \mathtt{C}, \mathtt{V}, \mathtt{T}).$$

$$r_{8.3} : \mathtt{repr}(\mathtt{T1}, \mathtt{C}, \mathtt{V}, \mathtt{T}) \leftarrow \mathtt{vtrain}(\mathtt{T}, \mathtt{C}, \mathtt{V}), \mathtt{C1} = \mathtt{C} - 1, \mathtt{repr}(\mathtt{Ta}, \mathtt{C1}, \mathtt{V1}, \mathtt{T}), \mathtt{rupt}(\mathtt{T1}, \mathtt{C1}, \mathtt{V1}, \mathtt{Ta}).$$

Assuming we want the rollup prefix table for count (Table 4), we can extract the node count using the aggregate $\mathtt{count}\langle \mathtt{TID} \rangle$ outside the recursion to derive the final table *myrupt* as follows.

$$r_{8.4} : \mathtt{myrupt}(\mathtt{T}, \mathtt{C}, \mathtt{V}, \mathtt{count}\langle \mathtt{TID} \rangle, \mathtt{Ta}) \leftarrow \mathtt{rupt}(\mathtt{T}, \mathtt{C}, \mathtt{V}, \mathtt{Ta}), \mathtt{repr}(\mathtt{Ta}, \mathtt{C}, \mathtt{V}, \mathtt{TID}).$$

In this example, the aggregate `count` could be transferred into recursion, but it would not save any computation time. However, if we want to further use anti-monotonic constraints like $(\mathtt{COUNT} \geqslant k)$[10] to prune many of the nodes from the prefix tree, then pushing count into recursion is a computationally efficient choice. Moreover, in the example, since the generation of the counts connected with the rollup prefix table is top-down, such lower-bound anti-monotonic constraints are *PreM*. The popular *a priori* constraint (Agrawal *et al.* 1994) used in frequent itemset mining is a well-known example that exploits *PreM*. It is also important to point out that generation of other aggregates like `max` and `min` on the rollup prefix table can be performed efficiently in a bottom-up manner when we have *PreM* constraints. In fact, the rollup computation can be stopped for (i) `max` when it fails the lower bound constraint and (ii) `min` when it fails the upper bound constraint.

Example 9 (*Computing length of the longest maximal pattern from a rollup prefix table*)

Many data mining applications extract condensed representations like maximal patterns (Giacometti *et al.* 2014) from rollup prefix-tree like structures (e.g., frequent-pattern tree or FP-tree (Han *et al.* 2000)). More recently, interesting mining applications have been developed, which depend on computing the length of the

[10] Often used in iceberg queries (Fang *et al.* 1998) and frequent itemset mining (Agrawal *et al.* 1994).

longest maximal pattern from a FP-tree[11] (Hu *et al.* 2008). The following Datalog program performs this task by using aggregates in recursion on the rollup prefix table for count `myrupt`.

$$r_{9.1} : \texttt{items}(\texttt{C}, \texttt{V}, \texttt{sum}\langle\texttt{Cnt}\rangle) \texttt{ <- } \texttt{myrupt}(\_, \texttt{C}, \texttt{V}, \texttt{Cnt}, \_).$$

$$r_{9.2} : \texttt{freqItems}(\texttt{C}, \texttt{V}) \texttt{ <- } \texttt{items}(\texttt{C}, \texttt{V}, \texttt{Cnt}), \texttt{Cnt} \geqslant \texttt{k}.$$

$$r_{9.3} : \texttt{len}(\texttt{T}, 0) \texttt{ <- } \texttt{myrupt}(\texttt{T}, \texttt{C}, \texttt{V}, \_, \_), \neg\texttt{myrupt}(\_, \_, \_, \_, \texttt{T}), \neg\texttt{freqItems}(\texttt{C}, \texttt{V}).$$

$$r_{9.4} : \texttt{len}(\texttt{T}, 1) \texttt{ <- } \texttt{myrupt}(\texttt{T}, \texttt{C}, \texttt{V}, \_, \_), \neg\texttt{myrupt}(\_, \_, \_, \_, \texttt{T}), \texttt{freqItems}(\texttt{C}, \texttt{V}).$$

$$r_{9.5} : \texttt{len}(\texttt{T}, \texttt{max}\langle\texttt{L}\rangle) \texttt{ <- } \texttt{len}(\texttt{TC}, \texttt{L1}), \texttt{myrupt}(\texttt{TC}, \_, \_, \_, \texttt{T}), \texttt{myrupt}(\texttt{T}, \texttt{C}, \texttt{V}, \_, \_), \neg\texttt{freqItems}(\texttt{C}, \texttt{V}), \texttt{L} = \texttt{L1}.$$

$$r_{9.6} : \texttt{len}(\texttt{T}, \texttt{max}\langle\texttt{L}\rangle) \texttt{ <- } \texttt{len}(\texttt{TC}, \texttt{L1}), \texttt{myrupt}(\texttt{TC}, \_, \_, \_, \texttt{T}), \texttt{myrupt}(\texttt{T}, \texttt{C}, \texttt{V}, \_, \_), \texttt{freqItems}(\texttt{C}, \texttt{V}), \texttt{L} = \texttt{L1} + 1.$$

$$r_{9.7} : \texttt{longest}(\texttt{max}\langle\texttt{L}\rangle) \texttt{ <- } \texttt{len}(\_, \texttt{L}).$$

Rules $r_{9.1}, r_{9.2}$ first compute singleton frequent items (i.e., items occurring above a threshold k), which are identified by the column number and its value. Since the longest maximal pattern occurs along the path from the leaf to the root of the prefix tree, rules $r_{9.5}, r_{9.6}$ recursively compute the maximum pattern length at each node from its descendants in a bottom-up manner from the leaves (selected by rules $r_{9.3}, r_{9.4}$) to the root. In addition to this, several other advanced analytics like iceberg queries (Fang *et al.* 1998), frequent itemset mining (Agrawal *et al.* 1994) and decision tree construction (Quinlan 1986) can be performed efficiently exploiting this rollup prefix table, as it has been pointed out in detail in Yang (2017).

5 Performance and scalability with multicore and distributed processing

Multicore and distributed systems were developed along different technology paths to provide two successful competing solutions to the problem of achieving scalability via parallelism. For a long time, Moore's law meant that programmers could virtually double the speed of their software by updating the hardware. But starting in 2005, *circa*, it became impossible to double transistor densities every two years. Since then therefore, computer manufacturers exploring alternative ways to increase performance developed the very successful computer line of *multicore processing* systems. Around the same time (2004–2005), the distributed processing approach to scalability used in cluster computing was developed by big users of BigData. This approach was first developed by database vendors, such as Teradata, and then popularized by web companies such as Google and Yahoo!, since they realized that *distributed processing* among their large clusters of shared-nothing computers provide an effective method to process their large and fast-growing data sets. The growing popularity of the distributed processing approach has been both the cause and the result of better programming support for parallel applications:

[11] A FP-tree is logically equivalent to a rollup prefix table.

for instance in MapReduce (Dean and Ghemawat 2004) users have only to provide a *map* and *reduce* program, while the system takes care of low level details such as data communication, process scheduling and fault tolerance. Finally, a major advance in usability was delivered by Apache Spark which provides higher level application programming interfaces (APIs) that have made possible the development of languages and systems supporting critical application areas, such as database applications written in SQL, graph applications using GraphX and data mining application suites. But Datalog can go beyond these advances by (i) providing unified declarative framework to support different applications, and (ii) achieving portability over different parallel systems. The significance of point (i) is underscored by the fact that BigDatalog was able to outperform GraphX on graph applications (Shkapsky *et al.* 2016), and the importance of point (ii) is demonstrated by the fact that while our Datalog applications will execute efficiently on both Apache Spark and multicore systems, the porting of parallel applications from the former platform to the latter can be quite challenging even for an expert programmer[12].

In the rest of the paper, we discuss the techniques used in ensuring that declarative programs expressed in Datalog have performance comparable to hand-written parallel programs on specialized domain-specific languages running on clusters of distributed shared-nothing computers (Shkapsky *et al.* 2016) and a multicore machine (Yang *et al.* 2017).

6 Datalog on Apache Spark

In this section, we provide a summary of BigDatalog–Spark (Shkapsky *et al.* 2016), a full Datalog language implementation on Apache Spark. BigDatalog–Spark supports relational algebra, aggregation and recursion, as well as a host of declarative optimizations. It also exploits the previously introduced semantic extensions for programs with *aggregation in recursion*. As a result, the Spark programmer can now implement complex analytics pipelines of relational, graph and machine learning tasks in a single language, instead of stitching together programs written in different APIs, i.e., Spark SQL (Armbrust *et al.* 2015), GraphX (Gonzalez *et al.* 2014) and MLlib.

6.1 Apache Spark

Apache Spark (Zaharia *et al.* 2012) is attracting a great deal of interest as a general platform for large-scale analytics, particularly because of its support for in-memory iterative analytics. Spark enhances the MapReduce programming model by providing a language-integrated Scala API enabling the expression of programs as *dataflows of second order transformations* (e.g., `map`, `filter`) on *resilient distributed datasets* (RDD) (Zaharia *et al.* 2012). An RDD is a distributed shared memory abstraction representing a partitioned dataset. RDDs are immutable, and transformations are

[12] Non-trivial optimization techniques such as those presented in Section 7 could be necessary in general.

coarse-grained and thus apply to all items in the RDD to produce a new RDD. RDDs can be explicitly cached by the programmer in memory or on disk at workers. RDDs provide fault tolerance by recomputing the sequence of transformations for the missing partition(s).

Once a Spark job is submitted, the scheduler groups transformations that can be pipelined into a single *stage*. Stages are executed *synchronously* in a topological order: a stage will not be scheduled until all stages it is dependent upon have finished successfully. Similar to MapReduce, Spark *shuffles* between stages to *repartition* outputs among the nodes of the cluster. Spark has libraries for structured data processing (Spark SQL), stream processing (Spark Streaming), machine learning (MLlib) and graph processing (GraphX).

Spark as a runtime for Datalog. Spark is a good candidate to support a Datalog compiler and Datalog evaluation; Spark is a general data processing system and provides the Spark SQL API (Armbrust *et al.* 2015). Spark SQL provides logical and physical relational operators, and Spark SQL's Catalyst compiler and optimizer supports the compilation and optimization of Spark SQL programs into physical plans. BigDatalog–Spark uses and extends Spark SQL operators, and also introduces operators implemented in the Catalyst framework so Catalyst planning features can be used on BigDatalog recursive plans.

BigDatalog–Spark is designed for general analytical workloads, and although we will focus much of the discussion and experiments on graph queries and recursive program evaluation, we do not claim that Spark is the best platform for graph workloads in general. In fact, BigDatalog can also be built into other general dataflow systems, including Naiad (Murray *et al.* 2013) and Hyracks (Borkar *et al.* 2011), and many of the optimization techniques presented in this section will also apply.

Challenges for Datalog on Spark. The following represent the main challenges with implementing Datalog on Spark:

(1) *Spark SQL supports acyclic plans:* Spark SQL lacks recursion operators, operators are designed for acyclic use, and the Catalyst optimizer plans non-recursive queries.

(2) *Synchronous scheduling:* Spark's synchronous stage-based scheduler requires unnecessary coordination for monotonic Datalog programs because monotonic Datalog programs are eventually consistent (Ameloot *et al.* 2011; Interlandi and Tanca 2015).

(3) *Memory utilization:* Each iteration of recursion will produce a new RDD to represent the updated recursive relation. If poorly managed, recursive applications on Spark can experience memory utilization problems.

6.2 BigDatalog–Spark

We highlight the features of BigDatalog–Spark with the help of the well-known transitive closure (Example 10) and same generation (Example 11) programs.

```
1  val bdCtx = new BigDatalogContext(sc)
2  val program = "database({arc(X:Integer, Y:Integer})."
3    + "tc(X,Y) <- arc(X,Y)."
4    + "tc(X,Y) <- tc(X,Z), arc(Z,Y)."
5  bdCtx.datalog(program)
6  bdCtx.datasource("arc", filePath)
7  val tc = bdCtx.query("tc(X,Y).")
8  val tcSize = tc.count()
```

Figure 1. BigDatalog–Spark program.

Example 10 (*Transitive closure (*TC*)*)

$$r_1 : \texttt{tc(X, Y)} \texttt{ <- } \texttt{arc(X, Y)}.$$
$$r_2 : \texttt{tc(X, Y)} \texttt{ <- } \texttt{tc(X, Z), arc(Z, Y)}.$$

r1 is an *exit rule* because it serves as a base case of the recursion. In r1, the arc predicate represents the edges of the graph – arc is a *base relation*. r1 produces a tc fact for each arc fact. r2 will recursively produce tc facts from the conjunction of previously produced tc facts and arc facts. The query to evaluate TC is of the form tc(X, Y). Last, this program uses a *linear* recursion in r2, since there is a single recursive predicate literal, whereas a *non-linear* recursion would have multiple recursive literals in its body. The number of iterations required to evaluate TC is, in the worst case, equal to the longest simple path in the graph.

Example 11 (*Same generation (*SG*)*)

$$r_1 : \texttt{sg(X, Y)} \texttt{ <- } \texttt{arc(P, X), arc(P, Y), X != Y}.$$
$$r_2 : \texttt{sg(X, Y)} \texttt{ <- } \texttt{arc(A, X), sg(A, B), arc(B, Y)}.$$

The exit rule r1 produces all X, Y pairs with the same parents (i.e., siblings) and the recursive rule r2 produces new X, Y pairs where both X and Y have parents of the same generation.

BigDatalog–Spark programs are expressed as Datalog rules, then compiled, optimized and executed on Spark. Figure 1 is the program to compute the size of the transitive closure of a graph using the BigDatalog–Spark API. The user first gets a BigDatalogContext (line 1), which wraps the SparkContext (sc) – the entry point for writing and executing Spark programs. The user then specifies a schema definition for base relations and program rules (lines 2–4). Lines 3 and 4 implement TC from Example 10. The database definition and rules are given to the BigDatalog–Spark compiler that loads the database schema into a relation catalog (line 5). Next, the data source (e.g., local or HDFS file path, or RDD) for the arc relation is provided (line 6). Then, the query to evaluate is given to the BigDatalogContext (line 7), which compiles it and returns an execution plan used to evaluate the query. As with other Spark programs, evaluation is lazy – the query is evaluated when count is executed (line 8).

Parallel semi-naïve evaluation on Spark. BigDatalog–Spark programs are evaluated using a parallel version of semi-naïve evaluation we call *parallel semi-naïve evaluation*

(PSN). PSN is an execution framework for a recursive predicate and it is implemented using RDD transformations. Since Spark evaluates synchronously, PSN will evaluate one iteration at a time; an iteration will not begin until all tasks from the previous iteration have completed.

The two types of rules for a recursive predicate – the *exit rules* and *recursive rules* – are compiled into separate *physical plans* (plans), which are then used in the PSN evaluator. Physical plans are composed of Spark SQL and BigDatalog–Spark operators that produce RDDs. The exit rules plan is first evaluated once, and then the recursive rules plan is repeatedly evaluated until a fixpoint is reached. Note that like the semi-naïve evaluation, PSN will also evaluate symbolically rewritten rules (e.g., $\texttt{tc(X, Y)} \leftarrow \delta\texttt{tc(X, Z)},\ \texttt{arc(Z, Y)}.$).

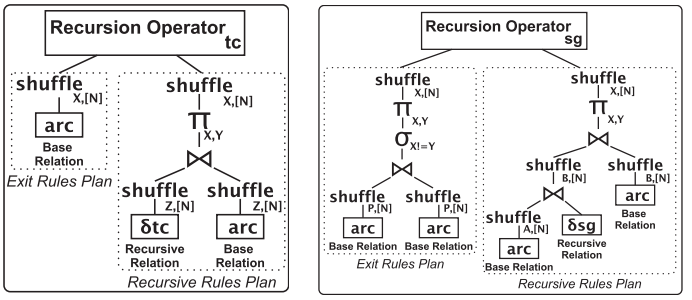
Algorithm 1 PSN evaluator with RDDs

```
1: delta = exitRulesPlan.toRDD().distinct()
2: all = delta
3: updateCatalog(all, delta)
4: do
5:     delta = recursiveRulesPlan.toRDD()
6:     .subtract(all).distinct()
7:     all = all.union(delta)
8:     updateCatalog(all, delta)
9: while (delta.count() > 0)
10: return all
```

Algorithm 1 is the pseudo-code for the PSN evaluator. The **exitRulesPlan** (line 1) and **recursiveRulesPlan** (line 5) are plans for the exit rules and recursive rules, respectively. We use `toRDD()` (lines 1 and 5) to produce the RDD for the plan. Each iteration produces two new RDDs – an RDD for the new results produced during the iteration (`delta`) and an RDD for all results produced thus far for the predicate (`all`). The `updateCatalog` (lines 3 and 8) stores new `all` and `delta` RDDs into a catalog for plans to access. The exit rule plan is evaluated first. The result is de-duplicated (`distinct`) (line 1) to produce the initial `delta` and `all` RDDs (line 2), which are used to evaluate the first iteration of the recursion. Each iteration is a new job executed by `count` (line 9). First, the **recursiveRulesPlan** is evaluated using the `delta` RDD from the previous iteration. This will produce an RDD that is set-differenced (`subtract`) with the `all` RDD (line 6) and de-duplicated to produce a new `delta` RDD. With lazy evaluation, the union of `all` and `delta` (line 7) from the previous iteration is evaluated prior to its use in `subtract` (line 6).

We have implemented PSN to cache RDDs that will be reused, namely `all` and `delta`, but we omit this from Algorithm 1 to simplify its presentation. Last, in cases of mutual recursion, when two or more rules belonging to different predicates reference each other (e.g., A ← B, B ← A), one predicate[13] will "drive" the recursion with PSN and the other recursive predicate(s) will be an operator in the driver's recursive rules plan. The "driver" predicate is determined from the

[13] Any of the mutually recursive predicates can be selected.



(a) TC (b) SG

Figure 2. PSN with SetRDD physical plans: (a) TC and (b) SG.

predicate connection graph, which is basically a dependency graph constructed by the compiler. The use of predicate connection graph is common in many Datalog system architectures like $\mathscr{LDL}$++ (Arni *et al.* 2003).

6.3 Optimizations

This section presents optimizations to improve the performance of BigDatalog–Spark programs. Details on the performance gain enabled by the discussed optimizations can be found in Tables 1–5 of the original BigDatalog–Spark paper (Shkapsky *et al.* 2016).

Optimizing PSN. As shown in Algorithm 1, PSN can be implemented with RDDs and standard transformations. However, using standard RDD transformations is inefficient because at each iteration the results of the recursive rules are set-differenced with the entire recursive relation (line 6 in Algorithm 1), which is growing in each iteration, and thus expensive data structures must be created for each iteration. We propose, instead, the use of *SetRDD*, which is a specialized RDD for storing distinct `Rows` and tailored for set operations needed for PSN. Each partition of a SetRDD is a set data structure. Although an RDD is intended to be immutable, we make SetRDD mutable under the union operation. The `union` mutates the set data structure of each SetRDD partition and outputs a new SetRDD comprised of these same set data structures. If a task performing union fails and must be re-executed, this approach will not lead to incorrect results because union is monotonic and facts can be added only once. Last, SetRDD transformations are implemented to not shuffle, and therefore the compiler must add shuffle operators to a plan. This approach allows for a simplified and generalized PSN evaluator.

Partitioning. An earlier research on Datalog showed that a good partitioning strategy (i.e., finding the arguments on which to partition) for a recursive predicate was important for an efficient parallel evaluation (Cohen and Wolfson 1989; Ganguly *et al.* 1990, 1992; Wolfson and Ozeri 1990). Since transferring data (i.e., communication) has a high cost in a cluster, we seek a partitioning strategy that limits shuffling. The default partitioning strategy employed by BigDatalog–Spark is to partition the recursive predicate on the *first argument*. Figure 2(a) is the plan for

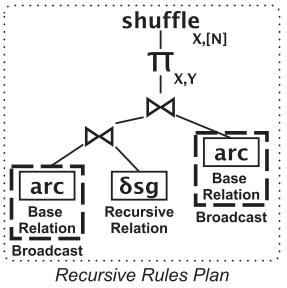


Figure 3. SG with broadcast joins.

Program 10 for PSN with SetRDD. With the recursive predicate (tc) partitioned on the first argument both the exit rule and recursive rule plans terminate with a shuffle operator.

In the plan in Figure 2(a), δtc requires shuffling prior to the join since it is not partitioned on the join key (Z) because the default partitioning is the first argument (X). However, if the default partitioning strategy was to use instead the second argument, the inefficiency with Figure 2(a) would be resolved but then other programs such as SG (plan shown in Fig. 2(b)) would suffer (δsg would require a shuffle prior to the join). Therefore, BigDatalog–Spark allows the user to define a recursive predicate's partitioning via configuration.

Join optimizations for linear recursion. By keeping the number of partitions static, a *shuffle join* implementing a linear recursion can have the non-recursive join input cached because the non-recursive inputs will not change during evaluation. This can lead to significant performance improvement since input partitions no longer have to be shuffled and loaded into lookup tables prior to the join in each iteration.

Instead of shuffle joins, each partition of a recursive relation can be joined with an entire relation (e.g., *broadcast join*). For either type of join, the non-recursive input is loaded into a lookup table. For a broadcast join, the cost of loading the entire relation into a lookup table is amortized over the recursion because the lookup table is cached and then reused in every iteration. Figure 3 shows a recursive rules plan for Example 11 (SG) with two levels of broadcast joins. In the event that a broadcast relation is used multiple times in a plan, as in Figure 3, BigDatalog–Spark will broadcast it once and share it among all broadcast join operators joining the relation.

Decomposable programs. Previous research on parallel evaluation of Datalog programs determined that some programs are *decomposable* and thus evaluable in parallel without redundancy (a fact is only produced once) and without processor communication or synchronization (Wolfson and Silberschatz 1988). Since mitigating the cost of synchronization and shuffling can lead to significant execution time speedup, enabling BigDatalog–Spark to support techniques for identifying and evaluating decomposable programs is desirable.

We consider a BigDatalog–Spark physical plan decomposable if the recursive rules plan has no shuffle operators. Example 10 (linear TC) is a decomposable program

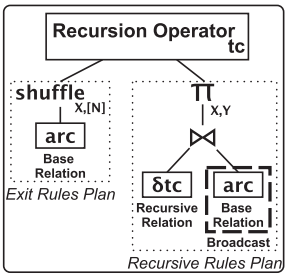


Figure 4. Decomposable TC plan.

(Wolfson and Silberschatz 1988); however, its physical plan shown in Figure 2(a) has shuffle operators in the recursive rules plan. Instead, BigDatalog–Spark can produce a decomposable physical plan for Example 10. First, `tc` will be partitioned by its first argument that divides the recursive relation so that each partition can be evaluated independently and without shuffling. Second, a broadcast join will be used which allows each partition of the recursive relation to join with the *entire* `arc` base relation. Figure 4 shows the decomposable physical plan for Example 10. Base relations are not pre-partitioned, therefore, the exit rules plan has a shuffle operator to repartition the `arc` base relation by `arc`'s first argument `X` into `N` partitions.

BigDatalog–Spark identifies decomposable programs via syntactic analysis of program rules using techniques presented in the *generalized pivoting* work (Seib and Lausen 1991). The authors of Seib and Lausen (1991) showed that the existence of a *generalized pivot set* for a program is a sufficient condition for decomposability and present techniques to identify generalized pivot set in arbitrary Datalog programs. When a BigDatalog–Spark program is submitted to the compiler, the compiler will apply the generalized pivoting solver to determine if the program's recursive predicates have generalized pivot set. If they indeed have, we now have a partitioning strategy and in conjunction with broadcast joins we can efficiently evaluate the program with these settings. For example, Example 10 has a generalized pivot set, which says to partition the `tc` predicate on its first argument. Note that this technique is enabled by using Datalog and it allows BigDatalog–Spark to analyze the program at the logical level. The Spark API alone is unable to provide this support since programs are written in terms of physical operations.

6.4 Experiments

In Shkapsky *et al.* (2016), we have tested BigDatalog–Spark over both synthetic and real-world datasets, and compared against other distributed Datalog implementations (e.g., Myria (Halperin *et al.* 2014) and SocialLite (Seo *et al.* 2013)), as well as hand-coded versions of programs implemented in Spark. The tests were executed using the TC, SG, CC, PYMK and MLM programs presented in Section 3 plus some additional ones. Here, we showcase a systems comparison using TC and SG (Fig. 5) and discuss results of scale-out and scale-up experiments (in Figs. 6 and 7, respectively). Each execution time reported in the figures is calculated by performing the same experiment five times, discarding the highest and lowest values

Table 6. *Parameters of synthetic graphs*

Name	Vertices	Edges	TC	SG
Tree11	71,391	71,390	805,001	2,086,271,974
Tree17	13,766,856	13,766,855	237,977,708	...
Grid150	22,801	45,300	131,675,775	2,295,050
Grid250	63,001	125,500	1,000,140,875	10,541,750
G5K	5,000	24,973	24,606,562	24,611,547
G10K	10,000	100,185	100,000,000	100,000,000
G10K-0.01	10,000	999,720	100,000,000	100,000,000
G10K-0.1	10,000	9,999,550	100,000,000	100,000,000
G20K	20,000	399,810	400,000,000	400,000,000
G40K	40,000	1,598,714	1,600,000,000	1,600,000,000
G80K	80,000	6,399,376	6,400,000,000	6,400,000,000

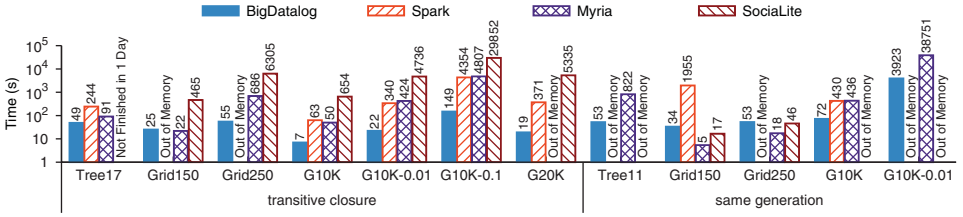


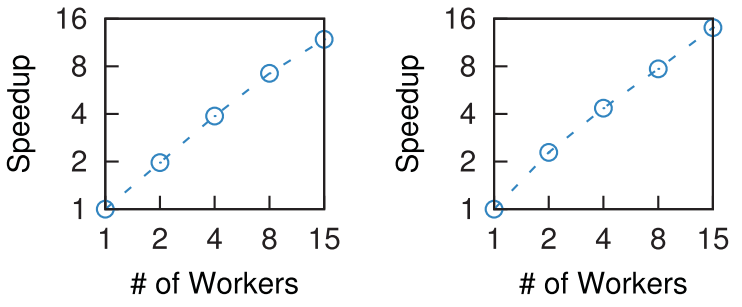
Figure 5. System comparison using TC and SG.

and taking the average of the remaining three values. The unit of time measurement is seconds.

Configuration. Our experiments were run on a 16-node cluster. Each node ran Ubuntu 14.04 LTS and had an Intel i7-4770 CPU (3.40 GHz, 4 core/8 thread), 32GB memory and a 1TB 7200 RPM hard drive. Nodes were connected with 1Gbit network. The BigDatalog–Spark implementation ran on Spark 1.4.0, and the file system is Hadoop 1.0.4.

Datasets. Table 6 shows the synthetic graphs used for the experiments of this section and of Section 7. Tree11 and Tree17 are trees of heights 11 and 17, respectively, and the degree of a non-leaf vertex is a random number between 2 and 6. Grid150 is a 151 by 151 grid, while Grid250 is a 251 by 251 grid. The Gn-p graphs are n-vertex random graphs (Erdős–Rényi model) generated by randomly connecting vertices so that each pair is connected with probability p. Gn-p graph names omitting p use default probability 0.001. Note that for these graphs, TC and SG are capable of producing result sets many orders of magnitude larger than the input dataset, as shown by the last two columns in Table 6.

Systems Comparison. For TC, BigDatalog–Spark uses Program 10 with the decomposed plan from Figure 4. For SG, BigDatalog uses Program 11 with broadcast joins (Fig. 3). We use equivalent programs in Myria and Socialite, and hand-optimized semi-naïve programs written in the Spark API, which are implemented to minimize shuffling. Figure 5 shows the evaluation time for all four systems.



(a) TC on G20K (b) SG on G10K

Figure 6. Scaling-out cluster size: (a) TC on G20K and (b) SG on G10K.

BigDatalog–Spark is the only system that finishes the evaluation for TC and SG on all graphs except SG on Tree17 since the size of the result is larger than the total disk space of the cluster. BigDatalog–Spark has the fastest execution time on six of the seven graphs for TC; on four of the graphs it outperforms the other systems by an order of magnitude. The BigDatalog–Spark plan only performs an initial shuffle of the dataset, and then evaluates the recursion without shuffling, and proves very efficient. In the case of Grid150, which is the smallest graphs used in this experiment in terms of both edges and queries output sizes, Myria outperforms BigDatalog–Spark both in TC and SG. This is explained as the evaluation requires many iterations, where each iteration performs very little work, and therefore the overhead of scheduling in BigDatalog–Spark takes a significant portion of execution time. However, as the data set becomes larger the superior scalability of BigDatalog–Spark comes into play enabling it to outperform all other systems on Grid250. In fact, Figure 5 shows that the execution time of BigDatalog–Spark on TC only grows to 2.2 times those of Grid150, whereas those of Myria and Socialite grow by more than one order of magnitude; from Grid150 to Grid250, BigDatalog–Spark also scales better on SG compared to the other systems. The Spark programs are not only affected by the overhead of scheduling and shuffling but also suffer memory utilization issues related to dataset caching and, therefore, ran out of memory for several datasets both in TC and SG.

Scalability. In this set of experiments, we use the G*n*-*p* graphs. Figure 6(a) shows the speedup for TC on G20K as the number of workers increases from 1 to 15 (all with one master) w.r.t. using only one worker, and Figure 6(b) shows the same experiment run for SG with G10K. Both figures show a linear speedup, with the speedup of using 15 workers as 12X and 14X for TC and SG, respectively.

The scaling-up results shown in Figure 7 were ran with the full cluster, i.e., one master and 15 workers. With each successively larger graph size, i.e., from G5K to G10K, the size of the transitive closure quadruples, but we do not observe a quadrupling of the evaluation time. Instead, evaluation time increases first less than 1.5X (G5K to G10K), then 3X (G10K to G20K), 6X (G20K to G40K), and 9X (G40K to G80K). Rather than focusing on the size of the TC w.r.t. execution time, the reason for the increase in execution time is explained by examining the results in Table 7.

Table 7. TC *Scaling-up Experiments Result Details*

Graph	Time braodcast(s)	TC	Generated facts	Generated facts/TC	Generated facts/second
G5K	4	24,606,562	122,849,424	4.99	30,712,356
G10K	6	100,000,000	1,001,943,756	10.02	166,990,626
G20K	17	400,000,000	7,976,284,603	19.94	469,193,212
G40K	119	1,600,000,000	50,681,485,537	31.68	425,894,836
G80K	1112	6,400,000,000	510,697,190,536	79.80	459,673,439

(Execution time not including the time to broadcast arc.)

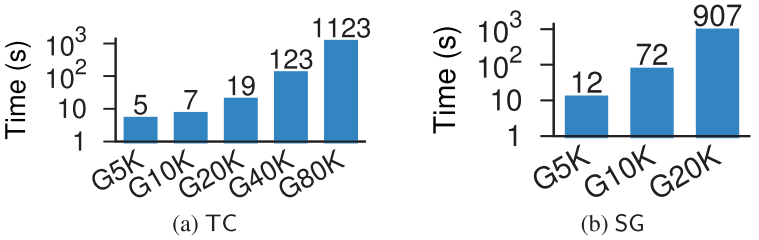
Table 8. SG *scaling-up experiments result details*

Graph	Time - broadcast(s)	SG	Generated Facts	Generated Facts / SG	Generated Facts / Sec.
G5K	11	24,611,547	612,891,161	24.90	55,717,378
G10K	71	100,000,000	10,037,915,957	100.38	141,379,098
G20K	905	400,000,000	159,342,570,063	398.36	176,069,138

(Execution time not including the time to broadcast arc)

Broadcasting the arc relation requires between 1s for G5K to 12s for G80K. Table 7 shows the execution time minus the time to broadcast arc, which is the total time the program required to actually evaluate TC. Table 7 also shows the number of generated facts, which is the number of total facts produced prior to de-duplication and is representative of the actual work the system must perform to produce the TC (i.e., HashSet lookups), the ratio between TC size and generated facts and the number of generated facts per second (time – broadcast time), which should be viewed as the evaluation throughput. These details help to explain why the execution times increase at a rate greater than the increase in TC size – the number of generated facts is increasing at a rate much greater than the increase in TC size. The last column shows that even with the increase in number of generated facts, BigDatalog–Spark still maintains good throughput throughout. Continuing, the first two graphs are too small to stress the system, but once the graph is large enough (e.g., G20K) the system exhibits a much greater throughput, which is stable across the larger graphs.

Table 8 displays the same details as Table 7 but for SG. Table 8 displays the execution time-minus the broadcast time of arc, the result set size, the number of



(a) TC (b) SG

Figure 7. Scaling-up on Random Graphs.

generated facts as well as statistics for the ratio of generated facts for each SG fact and generated fact per second of evaluation (throughput). With SG, the number of generated facts is much higher than we observe with TC, reflecting the greater amount of work SG requires. For example, on G10K and G20K SG produces 10X and 20X the number of generated facts, respectively, than TC produces. We also observe a much greater rate of increase in generated facts between graph sizes for SG compared to TC. For example, from G10K to G20K we see a 16X increase in generated facts for SG versus only an 8X increase for TC. For SG, we do not achieve as high a throughput as with TC, which is explained in part by the fact that SG requires shuffling, whereas our TC program evaluates purely in main memory after an initial shuffle.

7 Datalog on multicore systems: BigDatalog–MC

Multicore machines are composed by one or more processors, where each of them contains several cores on a chip (Venu 2011). Each core is composed of computation units and caches, while the main memory is commonly shared. While the individual cores do not necessarily run as fast as the highest performing single-core processors, they are able to obtain high performance by handling more tasks in parallel. In this paper, we will consider multicore processors implemented as a group of *homogeneous cores*, where the same computation logic is applied in a divide-and-conquer way over a partition of the input dataset.

Unfortunately, single-core applications do not get faster automatically on a multicore architecture with the increase of cores. For this reason, programmers are forced to write specific parallel logic to exploit the performance of multicore architectures. Next, we present the techniques used by BigDatalog–MC to enable the efficient parallel evaluation of Datalog programs over a shared-memory multicore machine with n processors.

7.1 Parallel bottom-up evaluation

We start with how BigDatalog–MC performs the parallel bottom-up evaluation of the transitive closure program TC in Example 10. We divide each relation into n partitions and we use the relation name with a superscript i to denote the ith partition of the relation. Each partition has its own storage for tuples, unique index and secondary indexes. Assuming that there are n workers that perform the actual query evaluation, and one coordinator that manages the coordination between the workers. Example 12 below shows a parallel evaluation plan for TC.

Example 12 (*Parallel bottom-up evaluation of* TC)
Let h be a hash function that maps a vertex to an integer between 1 and n. Both arc and tc are partitioned by the first column, i.e., $h(\mathtt{X}) = i$ for each $(\mathtt{X}, \mathtt{Y})$ in $\mathtt{arc}^i$ and $h(\mathtt{X}) = i$ for each $(\mathtt{X}, \mathtt{Y})$ in $\mathtt{tc}^i$. The parallel evaluation proceeds as follows.

(1) The ith worker evaluates the exit rule by adding a tuple $(\mathtt{X}, \mathtt{Y})$ to tc for each tuple $(\mathtt{X}, \mathtt{Y})$ in $\mathtt{arc}^i$.

(2) Once all workers finish Step (1), the coordinator notifies each worker to start Step (3).
(3) For each new tuple (X, Z) in $\texttt{tc}^i$ derived in the previous iteration, the ith worker looks for tuples of the form (Z, Y) in arc and adds a tuple (X, Y) to tc.
(4) Once all workers finish Step (3), the coordinator checks if the evaluation for tc is completed. If so, the evaluation terminates; otherwise, the evaluation starts from Step (3).

In Steps (1) and (3), each worker performs its task on one processor while the coordinator waits. Steps (2) and (4) serve as synchronization barriers.

In the above example, the ith worker only writes to $\texttt{tc}^i$ in Step (1), and it only reads from and writes to $\texttt{tc}^i$ in Step (3). Thus, $\texttt{tc}^i$ is only accessed by the ith worker. This property does not always hold in every evaluation plan of tc. For example, if we keep the current partitioning for arc but instead partition tc by its second column, then every worker could write to $\texttt{tc}^i$ in Step (3), and multiple write operations to $\texttt{tc}^i$ can occur concurrently; in this plan, we use a lock to ensure only one write operation to $\texttt{tc}^i$ is allowed at a time—a worker needs to acquire the lock before it writes to $\texttt{tc}^i$, and it releases the lock once the write operation completes.

In general, we use a lock to control the access to a partition if multiple read/write operations can occur concurrently. There are two types of locks: (i) an *exclusive lock* (x-lock) that allows only one operation at a time; and (ii) a *readers–writer lock* (rw-lock) that (a) allows only one write operation at a time, (b) allows concurrent read operations when no write operation is being performed and (c) disallows any read operation when a write operation is being performed. We use (i) an x-lock if there is no read operation and only multiple write operations can occur concurrently; (ii) an rw-lock if multiple read and write operations can occur concurrently since it allows for more parallelism than an x-lock.

We assume that every relation is partitioned using the same hash function h defined as follows:

$$h(x_1, \ldots, x_t) = \sum_{i=1}^{t} g(x_i) \bmod n,$$

where the input to h is a tuple of any arity t and g is a hash function with a range no less than n. Then, the key factor that determines whether locks are required during the evaluation is how each relation is partitioned, which is specified using *discriminating sets*. A discriminating set of a (non-nullary) relation R is a non-empty subset of columns in R. Given a discriminating set of a relation, we divide the relation into n partitions by the hash value of the columns that belong to the discriminating set. For each predicate p that corresponds to a base relation or a derived relation, let R be the relation that stores all tuples corresponding to facts about p in memory; we select a discriminating set of R that specifies the partitioning of R used in the evaluation of p. The collection of all the selected discriminating sets uniquely determines how each relation is partitioned. These discriminating sets can be arbitrarily selected as long as there is a unique discriminating set for each derived relation.

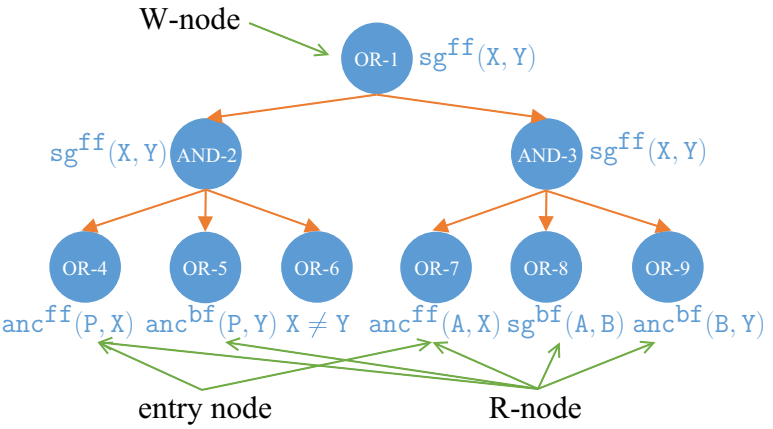


Figure 8. AND/OR tree of SG program in Example 11.

Example 13 (*Discriminating sets for the plan in Example 12*)
The discriminating sets for the two occurrences of `arc` are both $\{1\}$. Moreover, `tc` is a derived relation, and its discriminating set is $\{1\}$.

7.2 Parallel evaluation of AND/OR trees

The internal representation used by BigDatalog–MC to represent a Datalog program is an AND/OR tree (Arni *et al.* 2003). An OR node represents a predicate and an AND node represents the head of a rule. The root is an OR node. The children of an OR node (resp., AND node) are AND nodes (resp., OR nodes). Each node has a `getTuple` method that calls the `getTuple` methods of its children. Each successful invocation to the method instantiates the variables of one child (resp., all the children) and the parent itself for an OR node (resp., AND node). The program is evaluated by repeatedly applying the `getTuple` method upon its root until it fails. Thus, for an OR node, the execution (i) exhausts the tuples from the first child; (ii) continues to the next child; and (iii) fails when the last child fails. An OR node is an *R-node* if it reads from a base or derived relation with its `getTuple` method, while it is a *W-node* if it writes to a derived relation with its `getTuple` method. Finally, an OR node is an *entry node* if (i) it is a leaf, (ii) it is the first R-node among its siblings and (iii) none of its ancestor OR nodes has a left sibling (i.e., a sibling that appears before the current node) that has an R-node descendant or a W-node descendant.

Example 14 (*AND/OR tree of* SG)
Figure 8 shows the adorned AND/OR tree of the same generation program SG in Example 11, where (i) the text inside a node indicates its type and ID, e.g., "Or-1" indicates that the root is an OR node with ID 1, and (ii) the text adjacent to a node shows the corresponding predicate with its adornment (`b` or `f` in the ith position means the ith argument in a predicate p is bound or free when p is evaluated). Thus, Or-4, Or-5, Or-7, Or-8 and Or-9 are R-nodes, and Or-1 is a W-node. Or-4 and Or-7 are entry nodes in this program. Although Or-5 is an R-node, it is not an

entry node since it is not the first R-node among its siblings. Similarly, for OR-8 and OR-9.

In the parallel evaluation of an AND/OR tree with one coordinator and n workers, we create n copies of the same AND/OR tree, and assign the ith copy to the ith worker. The evaluation is divided into n disjoint parts, where the ith worker evaluates an entry node by instantiating variables with constants from the ith partition of the corresponding relation, while it has full access to all partitions of the corresponding relations for the remaining R-nodes. The parallel evaluation ensures the same workflow as the sequential pipelined evaluation by adding synchronization barriers in the nodes that represent recursion. For example, we create a synchronization barrier B, and add it to OR-1 of Figure 8 for every copy of the AND/OR tree. Now, the evaluation works as follows.

(1) Each worker evaluates the exit rule by calling AND-2.`getTuple` until it fails. A worker waits at B after it finishes.
(2) Once all n workers wait at B, the coordinator notifies each worker to start Step (3).
(3) Each worker evaluates the recursive rule by calling AND-3.`getTuple` until it fails. A worker waits at B after it finishes.
(4) Once all n workers wait at B, the coordinator checks if there are new tuples derived in `sg`. If so, the evaluation continues from Step (3); otherwise, the evaluation terminates.

7.3 Selecting a parallel plan

BigDatalog–MC uses a technique called *read/write analysis* (Yang *et al.* 2017) to help find the best discriminating sets to evaluate a program. For a given set of discriminating sets, the read/write analysis on an adorned AND/OR tree determines the actual program evaluation plan, including the type of lock needed for each derived relation, whether an OR node needs to acquire a lock before accessing the corresponding relation, and which partition of the relation an OR node needs to access when it accesses the relation through index lookups. The analysis performs a depth-first traversal on the AND/OR tree that simulates the actual evaluation to check each read or write operation performed by the ith worker. For each node N encountered during the traversal, the following three cases are possible:

(1) N is an entry node. In this case, set it as the current entry node; then, for each W-node that is an ancestor of N and is in the same stratum as N, determine whether the ith worker only writes to the ith partition of $R(p_w)$. This is done by checking if $p_e[\overline{X_j}] = p_w[\overline{X_k}]$[14], where p_e and p_w are the predicates associated with N and the W-node, respectively, and $\overline{X_j}$ and $\overline{X_k}$ are the corresponding discriminating sets.

[14] For a predicate p, $R(p)$ denotes the relation that stores all tuples corresponding to facts about p; $p[\overline{X}]$ denotes a tuple of arity $|\overline{X}|$ by retrieving the arguments in p whose positions belong to $\overline{X}$, and it is treated as a multiset of arguments when involved in equality checking.

(2) N is an R-node that reads from a derived relation. In this case, determine whether the ith worker only reads from the ith partition of $R(p_r)$ by checking if $\overline{X_k} \subseteq \overline{B}$ and $p_e[\overline{X_j}] = p_r[\overline{X_k}]$, where p_e and p_r are the predicates associated with the current entry node and N, respectively, $\overline{X_j}$ and $\overline{X_k}$ are the corresponding discriminating sets, and $\overline{B}$ is the set of positions for bound arguments in N.

(3) N is an R-node that reads from a base relation through a secondary index. In this case, determine whether the ith worker only needs to read from one partition of $R(p_r)$ instead of all the partitions by checking if $\overline{X_k} \subseteq \overline{B}$, where p_r is the predicate associated with N, $\overline{X_k}$ is the corresponding discriminating set, and $\overline{B}$ is the set of positions for bound arguments in N.

We can formulate the problem of determining the best discriminating sets for a given program as an optimization problem that minimizes the *cost of program evaluation*. This is equivalent to minimizing the overhead of program evaluation over the "ideal" plan in which all the constraints are satisfied. Now, for each OR node N in the AND/OR tree, its contribution to the overhead of program evaluation is denoted by $c(N)$, and its value is heuristically set as follows:

$$c(N) = \begin{cases} 3, \text{ if } N \text{ needs to acquire an r-lock (read lock) before performing an} \\ \quad \text{index lookup and condition } \overline{X_k} \subseteq \overline{B} \text{ is violated;} \\ 1, \text{ if } N \text{ needs to acquire a write lock before accessing the relation;} \\ 0, \text{ otherwise.} \end{cases}$$

Thus, the optimization problem reduces to finding an assignment that minimizes $\sum_N c(N)$, where N iterates over the set of OR nodes in the AND/OR tree. In BigDatalog–MC, this is achieved by enumerating all possible assignments using brute force, since the number of such valid assignments is totally tractable for most recursive queries of our interest. It is also important to take a closer look at the case where $c(N)$ equals three. There are two parts in the corresponding condition: first, N needs to acquire a read lock before performing an index lookup, and second, condition $\overline{X_k} \subseteq \overline{B}$ is violated. When $\overline{X_k} \subseteq \overline{B}$ is not true, this means we need to perform a lookup for each partition. This cost should be at least two, as there should be more than one partition during the parallel evaluation (otherwise, there is no need for parallelizing as there is only one partition and one processor). We also need to acquire a read lock for each lookup. However, we do not want to penalize this as much as acquiring a write lock, as acquiring a read lock is relatively less expensive. So, the contribution from the read lock is counted as one, and the overall cost is summed as three.

7.4 Experiments

Now, we introduce a set of experiments showcasing the performance of BigDatalog–MC compared to other (single and multicore) Datalog implementations, namely LogicBlox (Aref *et al.* 2015), DLV (Leone *et al.* 2006), clingo (Gebser *et al.* 2014) and SociaLite (Seo *et al.* 2013). Additional experiments and details can be found in Yang *et al.* (2017).

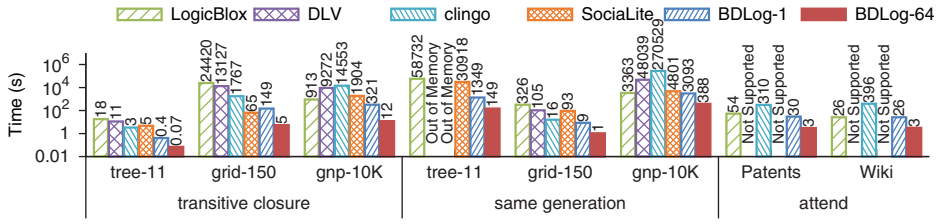


Figure 9. Query evaluation time of recursive queries.

Configuration. We tested the performance of the above systems on a machine with four AMD Opteron 6376 CPUs (16 cores per CPU) and 256GB memory (configured into eight NUMA regions). The operating system was Ubuntu Linux 12.04 LTS. We used LogicBlox 4.1.9 and CLINGO version 4.5.0. The version of DLV we used is a single-processor version[15], while for SociaLite we used the parallel version that was downloaded from `https://github.com/socialite-lang/socialite`.

System Comparison. Figure 9 compares the evaluation time of the five systems on TC, SG and ATTEND query. Bars for DLV and BDLog-1 show the evaluation time of DLV and BigDatalog–MC using one processor, while bars for LogicBlox, Clingo, SociaLite and BDLog-64 show the evaluation time of those systems over 64 processors. In our experiments, we observed both SociaLite and BigDatalog–MC had higher CPU utilization most of the time, as compared to LogicBlox and CLINGO, with the latter utilizing only one processor most of the time.[16]

When BigDatalog–MC is allowed to use only one processor, it always outperforms DLV and CLINGO. This comparison suggests that BigDatalog–MC provides a tighter implementation compared with the other two systems; specifically, we found that CLINGO, although a multicore Datalog implementation, spends most of the time on the grounder that utilizes only one processor.

Moreover, with only one processor, BigDatalog–MC outperforms or is on par with LogicBlox and SociaLite, while LogicBlox and SociaLite are allowed to use all 64 processors. Naturally, BigDatalog–MC always significantly outperforms LogicBlox and SociaLite when it uses all 64 processors. The performance gap between LogicBlox and BigDatalog–MC is largely due to the staged evaluation used by LogicBlox, which stores all the derived tuples in an intermediate relation, and performs deduplication or aggregation on the intermediate relation. For the evaluation that produces large amount duplicate tuples, such as TC on Grid150 and SG on Tree11, this strategy incurs a high space overhead, and the time spent on the deduplication, which uses only one processor, dominates the evaluation time. SociaLite instead uses an array of hash tables with an initial capacity of around 1,000 entries for a derived relation, whereas BigDatalog–MC uses an append-only

[15] The single-processor version of DLV is downloaded from `http://www.dlvsystem.com/files/dlv.x86-64-linux-elf-static.bin`. Although a parallel version is available from `http://www.mat.unical.it/ricca/downloads/parallelground10.zip`, it is either much slower than the single-processor version, or it fails since it is a 32-bit executable that does not support more than 4GB memory required by evaluation.

[16] These observations are obtained from the results of htop (see `https://hisham.hm/htop/`).

structure to store the tuples and a B+ tree to index the tuples. Although the cost of accessing a hash table is lower than that of a B+ tree, the design adopted by BigDatalog–MC allows a better memory allocation pattern as the relation grows. Such overhead is amplified when (i) multiple processors try to allocate memory at the same time, or (ii) the system has a high memory footprint.

Last, note that BigDatalog–MC achieves a greater speedup (the speedup of BDLog-64 over BDLog-1) for `TC` than `SG` and `ATTEND` since no lock is used in `TC`, while `SG` and `ATTEND` suffer from lock contention.

8 Related work

Datalog Semantics. Supporting aggregates in recursion is an old and difficult problem which has been the topic of much previous research work. Remarkably, previous approaches had primarily focussed on providing a formal semantics that could accommodate the non-monotonic nature of the aggregates. In particular, Mumick *et al.* (1990) discussed programs that are stratified w.r.t. aggregates operators and proved that a perfect model exists for these programs. Then, Kemp and Stuckey (1991) defined extensions of the well-founded semantics to programs with aggregation, and later showed that programs with aggregates might have multiple and counter-intuitive stable models. The notion of cost-monotonic extrema aggregates was introduced by Ganguly *et al.* (1995), using perfect models and well-founded semantics, whereas Greco *et al.* (1992) showed that their use to express greedy algorithms requires the do not-care non-determinism of the stable-model semantics provided by the choice construct. An approach to optimize programs with extrema was proposed by Ganguly *et al.* (1991), and a general optimization technique based on an early pruning of non-relevant facts was proposed by Sudarshan and Ramakrishnan (1991).

A general approach to deal with the four aggregates, min, max, count and sum, in a unified framework was proposed by Ross and Sagiv (1992) who advocated the use of semantics based on specialized lattices, different from set-containment, whereby each aggregate will then define a monotonic mapping in its specialized lattice. However, several limitations of this proposal were pointed out by Van Gelder (1993), including the assumption that cost arguments of atoms are functionally dependent from the other arguments. This is a property that does not hold in many applications and it also difficult to determine, since determining if a derived predicate satisfies a functional dependency is undecidable in general (Abiteboul and Hull 1988). In the following years, interest in aggregates for logic-based systems focussed on their use in the framework of answer-sets (Erdem *et al.* 2016), which is less conducive to BigData applications.

A renewed interest in BigData analytics brought a revival of Datalog for expressing more powerful data-intensive algorithms—including many that require aggregates in recursion. At UCLA, researchers first introduced the notion of monotonic sum and count (Mazuran *et al.* 2013b; Mazuran *et al.* 2013a), and then proposed the comprehensive solution that is described in this paper and covers all four basic aggregates along with efficient techniques for their efficient and scalable implementation.

Datalog Implementations. The Myria (Wang *et al.* 2015) runtime supports Datalog evaluation using a pipelined, parallel, distributed execution engine that evaluates graph of operators. Datasets are sharded and stored in PostgreSQL instances at worker nodes. SociaLite (Seo *et al.* 2013) is a Datalog language implementation for social network analysis. SociaLite programs are evaluated by parallel workers that use message passing to communicate. Both SociaLite and Myria support aggregation inside recursion focusing on their operational semantics. The lattice-based approach of Ross and Sagiv (1992) is proposed as the possible basis for a declarative semantics, but no approach on how to overcome its limitations is discussed. Furthermore, the advent of graphics processing units has recently led to Datalog implementations on graphics processing units for relational learning algorithms (Martínez-Angeles *et al.* 2016). Since the transfer of data between host and graphics processing unit memory incurs in significant cost, Datalog implementations on graphics processing units (Martınez-Angeles *et al.* 2014) optimize this cost through efficient memory management schemes.

Parallel Datalog Evaluation and Languages. Previous research on parallel evaluation of Datalog programs determined that some programs are evaluable in parallel without redundancy and without processor communication or synchronization (Wolfson and Silberschatz 1988). Such programs are called decomposable. Our parallel implementations identify decomposable programs via syntactic analysis of program rules using the generalized pivoting method (Seib and Lausen 1991). Others have also explored the idea of applying extensions of simple 0–1 laws on Datalog programs to derive at a parallelization plan that maximizes the expected performance (Lifschitz and Vianu 1998).

Many works produced over 20 years ago focussed on parallelization of bottom-up evaluation of Datalog programs (Zhang *et al.* 1995), however, they were largely of a theoretical nature. For instance, Van Gelder (1993) proposed a message passing framework for parallel evaluation of logic programs. Techniques to partition program evaluation efficiently among processors (Wolfson and Ozeri 1990), the trade-off between redundant evaluation and communication (Ganguly *et al.* 1990; Ganguly *et al.* 1992), and classifying how certain types of Datalog programs can be evaluated (Cohen and Wolfson 1989) were also studied. A parallel semi-naïve fixpoint has been proposed for message passing (Wolfson and Ozeri 1990) that includes a step for sending and receiving tuples from other processors during computation. The PSN used in this work applies the same program over different partitions and shuffle operators in place of processor communication. Parallel processing of recursive queries in particular is also a well-studied problem. One such example is Bell *et al.* (1991), where the recursive query is first transformed into a canonical form and then evaluated in a pipelined fashion.

Recently, Semantic Web reasoning systems dealing with Resource Description Framework (RDF) data has utilized this early research in parallel implementations of semi-naïve evaluation (Abiteboul *et al.* 1995) to handle recursive Datalog rules much like commercial systems as LogicBlox. One such system is RDFox (Motik *et al.* 2014), which is a main-memory, multicore RDF system that uses a specialized RDF indexing data structure to ensure largely lock-free concurrent

updates. It is also important to mention in this regard that, with the emergence of large Knowledge Graphs (Urbani *et al.* 2016), the Semantic Web community has significantly contributed to the ongoing research in Datalog reasoning. In fact, many reasoning systems encode RDF data, as represented in Knowledge Graphs, into ternary database predicates for writing elegant Datalog rules, which in turn, have to be efficiently evaluated. One such recent system is Vlog (Urbani *et al.* 2016), which exploits column-based memory layout along with selective caching of certain subquery results. However, Vlog is intrinsically sequential in nature and does not have a parallel or distributed implementation.

Among the distributed Datalog languages, it is noteworthy to mention *OverLog* (Loo *et al.* 2005; Condie *et al.* 2008), used in the P2 system to express overlay networks, and *NDlog* (Loo *et al.* 2006) for declarative networking. The *Bloom*^{L} (Conway *et al.* 2012) distributed programming language uses various monotonic lattices, also based on the semantics of Ross and Sagiv (1992), to identify program elements not requiring coordination. Bu *et al.* (2012) showed how XY-stratified Datalog can support computational models for large-scale machine learning, although no full Datalog language implementation on a large-scale system was provided.

Beyond Datalog: parallel execution of logic programs. In logic programming, programs are evaluated in a top-down fashion through *unification*. An extensive body of research was produced on parallel logic programming, dating back to 1981 (de Kergommeaux and Codognet 1994; Gupta *et al.* 2001). Two major approaches exists for parallelizing logic programs: the *implicit* approach assumes that the framework is able to parallelize the given input program automatically without any programmer intervention. Conversely, in the *explicit* case specific constructs are introduced into the source language to guide the parallel evaluation. The approach used in our BigDatalog systems is implicit parallelism where any input Datalog program is automatically parallelized by the runtime.

In implicit parallel logic programming, three main forms of parallelism exists: (i) *And-Parallelism* whereby multiple literals are evaluated concurrently; (2) *Or-Parallelism* where instead clauses are evaluated in parallel; and (3) *Unification Parallelism* in which the unification process is parallelized. Our parallel evaluation of Datalog programs is a form of Or-Parallelism where data is partitioned such that different rule instantiations are evaluated concurrently.

It is also important to note that the growth of Semantic Web data also propelled considerable research on large-scale reasoning on distributed frameworks like MapReduce (Dean and Ghemawat 2004). One such example is the WebPIE system (Urbani *et al.* 2012) that implements forward reasoning for RDFs over MapReduce framework. The key ideas originating from distributed MapReduce frameworks used for Semantic Web reasoning were also applied for description logic $\mathcal{EL}^+$ (Mutharaju *et al.* 2010) for $\mathcal{EL}^+$ ontology classifications. In this era of BigData, the Semantic Web community also led considerable research efforts towards large-scale non-monotonic reasoning of RDF data. One such paper is Tachmazidis *et al.* (2012), which proposed a MapReduce-based parallel framework for defeasible logic and predicates of any arity in presence of noisy data. In the same vein of large scale non-monotonic reasoning, the authors of Tachmazidis *et al.* (2014) proposed a similar data parallel

MapReduce framework for well-founded semantics computation through efficient implementations of joins and anti-joins.

9 Conclusion

By embracing the Horn-clause logic of Prolog but not its operational constructs such as the cut, Datalog researchers, 30 years ago, embarked in a significant expedition toward declarative languages in which logic alone rather than Logic+Control (Kowalski 1979) can be used to specify algorithms. Significant progress toward this ambitious goal was made in the 90s with techniques such as semi-naïve fixpoint and magic sets that support recursive Datalog programs by bottom-up computation and implementation techniques from relational DB systems. As described in Section 8, however, declarative semantics for algorithms that require aggregates in recursion largely remained an unsolved problem for this first generation of deductive DB systems. Moreover, Datalog scalability via parallelization was only discussed in papers, until recently when the availability of new parallel platforms and an explosion of interest in BigData renewed interest in Datalog and its parallel implementations on multicore and distributed systems.

In this paper, we have provided an in-depth description of the UCLA'a BigDatalog/DeAL project that is of significance because of its (i) historical continuity with first-generation Datalog systems ($\mathscr{LDL}$++ was supported and extended at UCLA for several years (Arni *et al.* 2003)), (ii) implementation on multiple platforms, with levels of performance that surpass those of competing Datalog systems, GraphX applications and even Apache Spark applications written in Scala and (iii) support for a wide range of declarative algorithms using the rigorous non-monotonic semantics for recursive programs with aggregates introduced in Zaniolo *et al.* (2017).

Furthermore, we believe that the use of aggregates in recursive rules made possible by *PreM* (Zaniolo *et al.* 2017) can lead to beneficial extensions in several application areas, e.g., knowledge discovery and data mining algorithms, and in related logic-based systems, including, e.g., those that use tabled logic programming (Swift and Warren 2012) and answer-sets (Erdem *et al.* 2016). Therefore, we see many interesting new topics deserving further investigation, suggesting that logic and databases remains a vibrant research area (Zaniolo *et al.* 2018), although many years have passed since it was first introduced (Minker *et al.* 2014).

Acknowledgement

We would like to thank the reviewers for the comments and suggested improvements.

References

Abiteboul, S. and Hull, R. 1988. Data functions, datalog and negation (extended abstract). In *Proc. of ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, June 1–3, 143–153.

Abiteboul, S., Hull, R. and Vianu, V., Eds. 1995. *Foundations of Databases: The Logical Level*, 1st ed., Addison-Wesley Longman Publishing, Boston, MA, USA.

Agrawal, R. *et al.* 1994. Fast algorithms for mining association rules. In *Proc. of 20th International Conference on Very Large Data Bases*, Vol. 1215, 487–499.

Ameloot, T. J., Neven, F. and Van den Bussche, J. 2011. Relational transducers for declarative networking. In *Proc. of 30th Principles of Database Systems (PODS)*, 283–292.

Aref, M. *et al.* 2015. Design and implementation of the logicblox system. In *Proc. of International Conference on Management of Data (SIGMOD)*. ACM, 1371–1382.

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 International Conference on Management of Data (SIGMOD)*, 1383–1394.

Arni, F., Ong, K., Tsur, S., Wang, H. and Zaniolo, C. 2003. The deductive database system LDL++. *Theory and Practice of Logic Programming 3*, 1, 61–94.

Bell, D. A., Shao, J. and Hull, M. E. C. 1991. A pipelined strategy for processing recursive queries in parallel. *Data & Knowledge Engineering 6*, 5, 367–391.

Borkar, V. R. *et al.* 2012. Declarative systems for large-scale machine learning. *IEEE Data Engineering Bulletin 35*, 2, 24–32.

Borkar, V. R., Carey, M. J., Grover, R., Onose, N. and Vernica, R. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. of 27th International Conference on Data Engineering (ICDE)*, 1151–1162.

Bu, Y., Borkar, V. R., Carey, M. J., Rosen, J., Polyzotis, N., Condie, T., Weimer, M. and Ramakrishnan, R. 2012. Scaling datalog for machine learning on big data. CoRR abs/1203.0160.

Cardoso, J. C., Baquero, C. and Almeida, P. S. 2009. Probabilistic estimation of network size and diameter. In *Proc. of 4th Latin-American Symposium on Dependable Computing (LADC'09)*. IEEE, 33–40.

Chimenti, D., O'Hare, A. B., Krishnamurthy, R., Tsur, S., West, C. and Zaniolo, C. 1987. An overview of the LDL system. *IEEE Data Engineering Bulletin 10*, 4, 52–62.

Cohen, S. and Wolfson, O. 1989. Why a single parallelization strategy is not enough in knowledge bases. In *Proc. of 8th Principles of Database Systems (PODS)*, 200–216.

Condie, T., Chu, D., Hellerstein, J. M. and Maniatis, P. 2008. Evita raced: Metacompilation for declarative networks. *Proceedings of the VLDB Endowment 1*, 1, 1153–1165.

Conway, N., Marczak, W. R., Alvaro, P., Hellerstein, J. M. and Maier, D. 2012. Logic and lattices for distributed programming. In *ACM Symposium on Cloud Computing (SOCC '12)*. San Jose, CA, USA, October 14–17.

Das, A. and Zaniolo, C. 2016. Fast lossless frequent itemset mining in data streams using crucial patterns. In *Proc. of SIAM International Conference on Data Mining*. Miami, Florida, USA, May 5–7, 576–584.

de Kergommeaux, J. C. and Codognet, P. 1994. Parallel logic programming systems. *ACM Computing Surveys 26*, 3, 295–336.

Dean, J. and Ghemawat, S. 2004. Mapreduce: Simplified data processing on large clusters. In *Proc. of 6th Symposium on Operating System Design and Implementation (OSDI)*, 137–150.

Erdem, E., Gelfond, M. and Leone, N. 2016. Applications of answer set programming. *AI Magazine 37*, 3, 53–68.

Faber, W., Pfeifer, G. and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence 175*, 1, 278–298.

Fang, M., Shivakumar, N., Garcia-molina, H., Motwani, R. and Ullman, J. D. 1998. Computing iceberg queries efficiently. In *Proc. of 24rd International Conference on Very Large Data Bases (VLDB)*, 299–310.

Ganguly, S., Greco, S. and Zaniolo, C. 1991. Minimum and maximum predicates in logic programming. In *Proc. of 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '91)*, 154–163.

Ganguly, S., Greco, S. and Zaniolo, C. 1995. Extrema predicates in deductive databases. *Journal of Computer and System Sciences 51*, 2, 244–259.

Ganguly, S., Silberschatz, A. and Tsur, S. 1990. A framework for the parallel processing of datalog queries. In *Proc. of International Conference on Management of Data (SIGMOD)*, 143–152.

Ganguly, S., Silberschatz, A. and Tsur, S. 1992. Parallel bottom-up processing of datalog queries. *Journal of Logic Programming 14*, 1, 101–126.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2014. Clingo= asp + control: Preliminary report. arXiv:1405.3694.

Gelfond, M. and Zhang, Y. 2014. Vicious circle principle and logic programs with aggregates. Theory and Practice of Logic Programming *14*, 4–5, 587–601. CoRR abs/1405.3637.

Giacometti, A., Li, D. H., Marcel, P. and Soulet, A. 2014. 20 years of pattern mining: A bibliometric survey. *SIGKDD Explorations Newsletter 15*, 1, 41–50.

Giannotti, F. and Manco, G. 2002. LDL-Mine: Integrating data mining with intelligent query answering. In *Proc. of Logics in Artificial Intelligence, European Conference, JELIA*, Cosenza, Italy, September, 23–26, 517–520.

Giannotti, F., Manco, G. and Turini, F. 2004. Specifying mining algorithms with iterative user-defined aggregates. *IEEE Transactions on Knowledge and Data Engineering 16*, 10, 1232–1246.

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J. and Stoica, I. 2014. Graphx: Graph processing in a distributed dataflow framework. In *Proc. of 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 599–613.

Greco, S., Zaniolo, C. and Ganguly, S. 1992. Greedy by choice. In *Proc. of 11th Symposium on Principles of Database Systems (PODS)*. ACM, 105–113.

Gupta, G., Pontelli, E., Ali, K. A., Carlsson, M. and Hermenegildo, M. V. 2001. Parallel execution of prolog programs: A survey. *ACM Transactions on Programming Languages and Systems 23*, 4, 472–602.

Halperin, D., de Almeida, V. T., Choo, L. L., Chu, S., Koutris, P., Moritz, D., Ortiz, J., Ruamviboonsuk, V., Wang, J., Whitaker, A., Xu, S., Balazinska, M., Howe, B. and Suciu, D. 2014. Demonstration of the myria big data management service. In *Proc. of International Conference on Management of Data (SIGMOD)*, Snowbird, UT, USA, June 22–27, 881–884.

Han, J., Pei, J. and Yin, Y. 2000. Mining frequent patterns without candidate generation. In *Proc. of International Conference on Management of Data (SIGMOD)*. ACM, 1–12.

Hu, T., Sung, S. Y., Xiong, H. and Fu, Q. 2008. Discovery of maximum length frequent itemsets. *Information Sciences 178*, 1, 69–87.

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

Kang, U., Tsourakakis, C. E., Appel, A. P., Faloutsos, C. and Leskovec, J. 2011. Hadi: Mining radii of large graphs. *ACM Transactions on Knowledge Discovery from Data 5*, 2, 8:1–8:24.

Kemp, D. B. and Stuckey, P. J. 1991. Semantics of logic programs with aggregates. In *Proc. of International Symposium on Logic Programming (ISLP)*. 387–401.

Kowalski, R. A. 1979. Algorithm = logic + control. *Communications of the ACM 22*, 7, 424–436.

Leone, N. *et al.* 2006. The DLV system for knowledge representation and reasoning. *Transactions on Computational Logic 7*, 3, 499–562.

Lewis, D. D. 1998. Naive (Bayes) at forty: The independence assumption in information retrieval. In *Proc. of 10th European Conference on Machine Learning (ECML '98)*. Springer-Verlag, London, UK, 4–15.

Lifschitz, S. and Vianu, V. 1998. A probabilistic view of datalog parallelization. *Theoretical Computer Science 190*, 2, 211–239.

Loo, B. T., Condie, T., Garofalakis, M. N., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T. and Stoica, I. 2006. Declarative networking: Language, execution and optimization. In *Proc. of International Conference on Management of Data (SIGMOD)*. ACM, 97–108.

Loo, B. T., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T. and Stoica, I. 2005. Implementing declarative overlays. In *Proc. of 20th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 75–90.

Martinez-Angeles, C. A., Dutra, I. and Costa, V. S. 2014. A datalog engine for GPUs. *Declarative Programming and Knowledge Management*, Springer, 152–168.

Martínez-Angeles, C. A., Wu, H., Dutra, I., Costa, V. S. and Buenabad-Chávez, J. 2016. Relational learning with GPUs: Accelerating rule coverage. *International Journal of Parallel Programming 44*, 3, 663–685.

Matula, D. W. and Beck, L. L. 1983. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM 30*, 3, 417–427.

Mazuran, M., Serra, E. and Zaniolo, C. 2013a. A declarative extension of horn clauses, and its significance for datalog and its applications. *Theory and Practice of Logic Programming 13*, 4–5, 609–623.

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

Minker, J., Seipel, D. and Zaniolo, C. 2014. Logic and databases: A history of deductive databases. In *Computational Logic*, Elsevier, 571–627.

Mitchell, T. M. 1997. *Machine Learning*. McGraw-Hill, Boston, MA.

Morris, K. A., Ullman, J. D. and Gelder, A. V. 1986. Design overview of the nail! system. In *Proc. of 3rd International Conference on Logic Programming, Imperial College of Science and Technology*. London, UK, July 14–18, 554–568.

Motik, B., Nenov, Y., Piro, R., Horrocks, I. and Olteanu, D. 2014. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *Proc. of 28th AAAI Conference on Artificial Intelligence (AAAI'14)*. AAAI Press, 129–137.

Mumick, I. S., Pirahesh, H. and Ramakrishnan, R. 1990. The magic of duplicates and aggregates. In *Proc. of 16th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers, 264–277.

Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P. and Abadi, M. 2013. Naiad: A timely dataflow system. In *Proc. of 24th Symposium on Operating Systems Principles (SOSP)*, 439–455.

Mutharaju, R., Maier, F. and Hitzler, P. 2010. A mapreduce algorithm for SC. In *Proc. of 23rd International Workshop on Description Logics (DL'10)*, 456.

Pelov, N., Denecker, M. and Bruynooghe, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming 7*, 3, 301–353.

Przymusinski, T. C. 1988. Perfect model semantics. In *Proc. of International Conference and Symposium on Logic Programming (ICLP/SLP)*, 1081–1096.

Quinlan, J. R. 1986. Induction of decision trees. *Machine Learning 1*, 1, 81–106.

Ramakrishnan, R., Srivastava, D. and Sudarshan, S. 1992. CORAL – Control, relations and logic. In *Proc. of 18th International Conference on Very Large Data Bases*, August 23-27. Vancouver, Canada, 238–250.

Ross, K. A. and Sagiv, Y. 1992. Monotonic aggregation in deductive databases. In *Proc. of 11th Symposium on Principles of Database Systems (PODS)*. ACM, 114–126.

Seib, J. and Lausen, G. 1991. Parallelizing datalog programs by generalized pivoting. In *Proc. of 10th Symposium on Principles of Database Systems (PODS)*, 241–251.

Seo, J., Guo, S. and Lam, M. S. 2013. SociaLite: Datalog extensions for efficient social network analysis. In *Proc. of International Conference on Data Engineering (ICDE'13)*. IEEE, 278–289.

Seo, J., Park, J., Shin, J. and Lam, M. S. 2013. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment 6*, 14, 1906–1917.

Shin, K., Eliassi-Rad, T. and Faloutsos, C. 2016. Corescope: Graph mining using k-core analysis – Patterns, anomalies and algorithms. In *Proc. of 16th International Conference on Data Mining (ICDM)*. IEEE, 469–478.

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 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1135–1149.

Shkapsky, A., Zeng, K. and Zaniolo, C. 2013. Graph queries in a next-generation datalog system. *Proceedings of the VLDB Endowment 6*, 12, 1258–1261.

Simons, P., Niemelä, I. and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 1–2, 181–234.

Son, T. C. and Pontelli, E. 2007. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming 7*, 3, 355–375.

Sudarshan, S. and Ramakrishnan, R. 1991. Aggregation and relevance in deductive databases. In *Proc. of 17th International Conference on Very Large Data Bases (VLDB)*, 501–511.

Swift, T. and Warren, D. S. 2010. Tabling with answer subsumption: Implementation, applications and performance. In *Proc. of European Workshop on Logics in Artificial Intelligence (JELIA)*. 300–312.

Swift, T. and Warren, D. S. 2012. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming 12*, 1–2, 157–187.

Tachmazidis, I., Antoniou, G. and Faber, W. 2014. Efficient computation of the well-founded semantics over big data. *Theory and Practice of Logic Programming 14*, 4–5, 445–459.

Tachmazidis, I., Antoniou, G., Flouris, G., Kotoulas, S. and McCluskey, L. 2012. Large-scale parallel stratified defeasible reasoning. In *Proc. of 20th European Conference on Artificial Intelligence*. IOS Press, 738–743.

Tsur, S. 1991. Deductive databases in action. In *Proc. of 10th Symposium on Principles of Database Systems (PODS '91)*. . ACM, New York, NY, USA, 142–153.

Urbani, J., Jacobs, C. J. and Krötzsch, M. 2016. Column-oriented Datalog Materialization for large knowledge graphs. In *Proc. of 30th Conference on Artificial Intelligence (AAAI)*, 258–264.

Urbani, J., Kotoulas, S., Maassen, J., Van Harmelen, F. and Bal, H. 2012. Webpie: A web-scale parallel inference engine using MapReduce. *Web Semantics: Science, Services and Agents on the World Wide Web 10*, 59–75.

Vaghani, J., Ramamohanarao, K., Kemp, D. B., Somogyi, Z., Stuckey, P. J., Leask, T. S. and Harland, J. 1994. The Aditi deductive database system. *VLDB Journal 3*, 2, 245–288.

Van Gelder, A. 1993. Foundations of aggregation in deductive databases. In *Proc. of International Conference on Deductive and Object-Oriented Databases*. Springer, 13–34.

Venu, B. 2011. Multi-core processors – An overview. CoRR abs/1110.3535.

Wang, J., Balazinska, M. and Halperin, D. 2015. Asynchronous and fault-tolerant recursive Datalog evaluation in shared-nothing engines. *Proceedings of the VLDB Endowment 8*, 12, 1542–1553.

Wolfson, O. and Ozeri, A. 1990. A new paradigm for parallel and distributed rule-processing. In *Proc. of International Conference on Management of Data (SIGMOD)*, 133–142.

Wolfson, O. and Silberschatz, A. 1988. Distributed processing of logic programs. In *Proc. of International Conference on Management of Data (SIGMOD)*, 329–336.

Yang, M. 2017. *Declarative Languages and Scalable Systems for Graph Analytics and Knowledge Discovery*. Ph.D. thesis, UCLA.

Yang, M., Shkapsky, A. and Zaniolo, C. 2015. Parallel bottom-up evaluation of logic programs: DeALS on shared-memory multicore machines. In *Technical Communications of ICLP*, Cork, Ireland.

Yang, M., Shkapsky, A. and Zaniolo, C. 2017. Scaling up the performance of more powerful datalog systems on multicore machines. *VLDB Journal 26*, 2, 229–248.

Yang, M. and Zaniolo, C. 2014. Main memory evaluation of recursive queries on multicore machines. In *Proc. of IEEE International Conference on Big Data*, 251–260.

Yu, Y., Gunda, P. K. and Isard, M. 2009. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *Proc. of 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 247–260.

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 9th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

Zaniolo, C., Yang, M., Interlandi, M., Das, A., Shkapsky, A. and Condie, T. 2017. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *Theory and Practice of Logic Programming 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 12th Alberto Mendelzon International Workshop on Foundations of Data Management*, Cali, Colombia, May 21–25.

Zhang, W., Wang, K. and Chau, S.-C. 1995. Data partition and parallel evaluation of datalog programs. *IEEE Transactions on Knowledge and Data Engineering 7*, 1, 163–176.