Title
A Declarative Language for Advanced Analytics and its Scalable Implementation

Author
Shkapsky, Alexander Philip

Publication Date
2016

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

A Declarative Language for Advanced Analytics and its Scalable Implementation

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Alexander Philip Shkapsky

2016

ABSTRACT OF THE DISSERTATION

A Declarative Language for Advanced Analytics and its Scalable Implementation

by

Alexander Philip Shkapsky

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2016

Professor Carlo Zaniolo, Chair

Advanced analytics are used to discover hidden patterns and trends in massive datasets. Great strides have been made by researchers to provide computational models, systems and accompanying languages for analytics. However, there is still a dire need for highly expressive declarative languages that enable the compilation, optimization and evaluation of advanced analytics over massive datasets. Specifically, a language for analytics needs (i) to support the expression of analytics over multiple data models (ii) to provide high-level declarative constructs enabling system optimizations, and (iii) be conducive for iterative or recursive evaluation.

In this dissertation, we propose an expressive Datalog language for advanced analytics, and compilation and optimization techniques for its efficient evaluation on systems designed for iterative execution. Specifically, this dissertation makes two main contributions:

(i) We develop and demonstrate a next generation Datalog System - the **De**ductive **A**pplication **L**anguage **S**ystem ($DeALS$). To extend the range of analytics supported in $DeALS$, we add support for aggregation in recursion into our logic-based language. We propose the design and implementation of several monotonic aggregates that can be used in recursive Datalog rules and evaluated efficiently using our novel optimization techniques. We demonstrate the effectiveness of these aggregates and conduct an experimental comparison with other Datalog systems and determine that $DeALS$ combines superior generality with superior performance.

(ii) We design and implement BigDatalog, a Datalog system on Apache Spark, for large-scale advanced analytics. We implement BigDatalog for efficient distributed evaluation and to utilize communication-reduction techniques during evaluation. We propose compilation and optimization techniques, as well as job scheduling techniques, to support efficiently the evaluation of *DeAL* programs on Spark. We conduct an experimental comparison with other state-of-the-art large-scale Datalog systems and demonstrate the efficacy of our techniques and effectiveness of our Spark extensions in supporting Datalog-based analytics.

The dissertation of Alexander Philip Shkapsky is approved.

Tyson Condie

Todd D. Millstein

Vwani P. Roychowdhury

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

2016

For my wife

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I give my sincerest thanks and appreciation to my advisor Professor Carlo Zaniolo. I am grateful for his time and patience. I am grateful for his insights on my work and for his thoughtful advice on how to be a better researcher and better market my research.

I would like to thank my committee, Professor Tyson Condie, Professor Todd Millstein, and Professor Vwani Roychowdhury, for their advice and suggestions on my research.

I would like to thank my fellow students. I am sure I am forgetting some who helped me along the way, but this list includes Muhao Chen, Hsuan Chiu, Ariyam Das, Shi Gao, Jiaqi Gu, Matteo Interlandi, Nikolay Laptev, Neng Lu, Giuseppe Mazzeo, Hamid Mousavi, Barzan Mozafari, Jin Wang, Mohan Yang, Kai Zeng.

I thank my wife Tricia for her love and patience during my years as a student. Without her strength and support I would not been have able to complete this dissertation.

VITA

2001	Bachelor of Science in Commerce, Emphasis in Operations and Management Information Systems, Santa Clara University, Santa Clara, California.
2001-2002	Junior Software Engineer, Data Systems & Solutions, San Diego, California.
2003-2004	Software Developer, First American Default Technologies, Anaheim, California.
2004-2005	Lead Developer, First American Default Technologies, Anaheim, California.
2005-2007	Senior Software Engineer, BDS Marketing, Irvine, California.
2008-2009	Senior Software Developer, Vendor Resource Management, Pomona, California.
2009	Master of Science in Computer Science, UCLA.
2009-2011	Lead Developer, Vendor Resource Management, Pomona, California.
2011-2012	Graduate Student Researcher, UCLA International Institute, UCLA.
2012	Software Engineer Intern, Commission Junction, Los Angeles, California.
2012-2015	Graduate Student Researcher, Computer Science Department, UCLA.

PUBLICATIONS

Mohan Yang, **Alexander Shkapsky**, Carlo Zaniolo. Parallel Bottom-Up Evaluation of Logic Programs: DeALS on Shared-Memory Multicore Machines. In Proceedings of the Technical Communications of the 31st International Conference on Logic Programming. (**ICLP**), Cork, Ireland, August 31 - September 4, 2015.

Alexander Shkapsky, Mohan Yang, Carlo Zaniolo. Optimizing recursive queries with monotonic aggregates in DeALS. In Proceedings of the 31st IEEE International Conference on Data Engineering (**ICDE**), Seoul South Korea, April 13-17, 2015.

Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. Graph Queries in a Next Generation Datalog System. Demo. In Proceedings of the 39th International Conference on Very Large Data Bases (**VLDB**), Rival del Garda, Trento, August 26-30, 2013.

CHAPTER 1

Introduction

The high velocity of the modern world requires accurate, timely and sophisticated analytics to support informed decision making, better allocation of resources and identification of new opportunities. In fact, analytics is responsible for many of the advanced applications we use daily. Analytics is a broad term and includes tasks from data analysis, data mining and machine learning. Using web analytics to track user visits and page views, websites can better allocate resources and balance load across their clusters of web servers. Network intrusion detection systems need to quickly process network and server logs to identify unwanted visitors. Algorithms such as PageRank [PBM99] are used to identify the most influential members in a social network graph or important websites in a web graph.

The development of advanced analytics is an iterative process that requires tuning a pipeline of subtasks and can involve a variety of systems and tools. Common subtasks include i) filtering, joining and aggregating multiple datasets, ii) designing, implementing and testing complex learning models and iii) ensuring the application can properly scale. Moreover, a single analytics application can require elements of multiple data models, such as both relational and graph datasets.

Big Data is a rich application area for analytics due in part to the massive and continually growing datasets and the variety of data available. Both industry and academia have proposed many scalable systems and computational models for Big Data analytics. Google's MapReduce [DG04], and its open source version Hadoop MapReduce [had15], have led the charge by providing practitioners with a powerful framework for distributed parallel processing of large-scale batch jobs. Platforms supporting a larger or more general operator set than MapReduce [BEH10, IBY07]

or extending the MapReduce framework with iterative constructs [BHB12, ELZ10, pro15, RPB13, SKH12, ZGG12, ZGG11] have also been presented. Parallel RDBMS [CDD09, FPC09, LFV12], designs based on parallel database systems [BCG11] and hybrid MapReduce/RDBMS models [ABH10, YYT10] have also emerged. Distributed in-memory designs for low-latency data analysis and iterative machine learning [PL10, ZCD12] have appeared. For large-scale graph analytics and graph machine learning, systems for both parallel [KBG12, LGK10] and distributed [gir15, GLG12, LBG12, MAB10, SWL13, TBC13] processing have been proposed.

However, even with these powerful systems available, advanced analytics still requires better support from powerful high-level programming languages with advanced built-in constructs backed by declarative semantics that are amenable to parallelization on multiple platforms. We identify three language challenges for advanced analytics:

First, as the needs for analytics applications can vary greatly from acyclic batch processing over relational data to long-running iterative and recursive workflows on graphs, a language should be *expressive enough to support both relational and graph data models*. For example, using a MapReduce system for many types of graphs processing is inefficient, whereas the vertex-centric models used by some graph-based systems are not suitable for many other types of analytics; it is also difficult to express graph queries using SQL. Often programmers are plagued with unnecessary code complexity from having to compensate for a computational model that quickly becomes suboptimal in terms of execution performance and scalability. Furthermore, this results in the programmer having to stitch together application subcomponents written in different APIs or executed by different systems. Supporting multiple data models at the language level increases logical data independence, reduces portability concerns and helps mitigate the need to migrate from one language or low-level API to another, which is often a major engineering effort.

Second, a language should provide *high-level declarative constructs enabling system optimization*. A noticeable construct lacking from most languages proposed for analytics is recursion. Recent work has proposed moving analytics into the RDBMS using built-in or user-defined function for analytics [HRS12, FKR12], but these approaches inherit the RDBMS's poor support for recursion. Although domain specific languages (DSL) such as ScalOps [WCR11] are a step in the

right direction, other language integrated APIs [YIF08, ZCD12] embedded in another high-level language inherit support for iteration through the language's control flow (e.g., C# or Scala). By enabling the user to specify *what* they want to do and tasking the system with *how* to do it, the system is given the best opportunity to provide optimized execution plans. We see this in declarative languages, such as SQL and Datalog, as optimizers can more easily identify declarative constructs and apply well-known optimization techniques.

Lastly, to enable the efficient support of sophisticated analytics, which include data mining and machine learning algorithms, the language must be evaluated by a *system that efficiently supports iteration*. One approach has been to port existing languages, such as SQL, to large-scale platforms for batch processing. However, the ported language is limited to efficiently supporting only the applications the platform can support efficiently. For example, high-level languages [BEG11, GNC09, TSJ09] and frameworks [cas15a, cas15b, sca15] for Hadoop efficiently support pipelines of acyclic batch jobs but poorly support iterative jobs. Another example is how RDBMS and their compiler optimizers targeting acyclic executions poorly support recursive queries. A system designed for iteration and/or recursion can more naturally and efficiently support a wide-range of complex analytics tasks.

1.1 Overview

In this dissertation, we made the following contributions towards the challenges outlined above.

1.1.1 Deductive Application Language System

We have developed the **De**ductive **A**pplication **L**anguage **S**ystem ($DeALS$) for analytics. $DeALS$ builds on earlier Deductive Database technology [AOT03] to efficiently support $DeAL$, a DATALOG language that supports recursion, aggregation, negation and advanced constructs including user-defined aggregates and non-monotonic aggregation in recursion via XY-stratified Datalog [ZAO93]. $DeALS$ supports the expression and evaluation of a wide range of complex applications including relational, aggregate, graph and recursive queries – all within a single unified declarative

language.

1.1.2 Aggregation in Recursion

Aggregates in recursive queries are essential in many important applications including shortest paths computations, link and graph structure analysis and bill of materials queries. To enable the expression of these and many other analytics programs in our language, we design and efficiently implement a set of monotonic aggregates that are based on recent theoretical results [MSZ13a, MSZ13b] that use the set-containment semantics of standard DATALOG, and therefore can be used in recursive rules and evaluated efficiently. We provide novel optimization techniques to evaluate recursive rules with monotonic aggregates efficiently. We conduct an experimental comparison with other DATALOG systems which showed that our implementation in $DeALS$ indeed provides superior performance, especially on sparse graphs.

1.1.3 BigDatalog

To support complex analytics on a scalable runtime over large datasets, we design and implement BigDatalog, a Datalog system on Apache Spark. We propose efficient compilation and optimization techniques specifically for Spark to enable the efficient evaluation of BigDatalog programs, including recursive programs with monotonic aggregates. We propose system-level optimizations in Spark to increase the efficiency of recursive query evaluation. We conduct an experimental comparison with (i) other large-scale state-of-the-art DATALOG systems, (ii) native Spark programs, and (iii) Spark's graph processing module GraphX. The results show that BigDatalog outperforms the other systems on classical recursive queries such as transitive closure, and also outperforms on many important analytics queries on large datasets.

1.1.4 Outline

The remainder of this dissertation is organized as follows. Chapter 2 reviews Datalog preliminaries. Chapter 3 presents our contributions toward $DeAL$ and its system ($DeALS$). Chapter 4 presents

the design and implementation of monotonic aggregates that can be used in recursive Datalog programs. Chapter 5 presents BigDatalog. We conclude and present future work in Chapter 6.

CHAPTER 2

Background: Datalog

A DATALOG program is a finite set of rules. A *rule* r has the form $h \leftarrow b_1, \ldots, b_n$, where h is the $head$ of r and $b_1, \ldots, b_n$ is the $body$. h and each b_i are *literals* with the form $p_i(t_1, \ldots, t_j)$ where p_i is a *predicate* and $t_1, \ldots, t_j$ are *terms* which can be *constants*, *variables* or *functions*. We say r is a rule of predicate h, unless it has an empty body, and then it is a $fact$. The comma separating literals in a body is a logical conjunction (AND). A successful assignment of all variables in the body produces a fact for the head's predicate. Predicates are also considered *relations* and *w.l.o.g.* throughout this dissertation we will use the terms predicate and relation interchangeably. A *query* indicates the desired predicate to evaluate. As a convention, predicate and function names begin with lower case letters, and variable names begin with upper case letters.

Datalog by Example. Program 1 is *Transitive Closure* (TC), the quintessential DATALOG program. TC recursively produces all pairs of vertices that are connected by some path in a graph.

Program 1. *Transitive Closure*

$$r1.\texttt{tc}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{arc}(\texttt{X}, \texttt{Y}).$$
$$r2.\texttt{tc}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{tc}(\texttt{X}, \texttt{Z}),\ \texttt{arc}(\texttt{Z}, \texttt{Y}).$$

Program 1 is explained as follows. *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* (i.e., a table in RDBMS). *r1* produces a `tc` fact for each `arc` fact. *r2* is a *recursive rule* since it has the `tc` predicate in both its head and body. *r2* will recursively produce `tc` facts from the conjunction (i.e., equi-join in relational terminology) of previously produced `tc` facts and `arc` facts. The query to evaluate TC is of the form `tc(X,Y)`. Lastly, this program uses a *linear* recursion in *r2*, since there is a single recursive predicate literal, whereas a *non-linear* recursion will have multiple

recursive literals in its body. The number of iterations required to evaluate Program 1 is equal to the longest simple path in the graph.

2.1 Datalog Evaluation

DATALOG is a declarative language and therefore rules are independent of the operators used to implement them (e.g., type of join used). Furthermore, rules are independent of the particular evaluation order and technique used as long as the monotonic *w.r.t.* set-containment[1], and least fixpoint, semantics of DATALOG is maintained. Lastly, the order of literals in a rule body provides no semantic meaning and most implementations, including this work, evaluate literals in a left-to-right fashion.

A *naïve* evaluation of Program 1 will execute *r1* and then repeatedly evaluate *r2*, joining `arc` facts with already discovered `tc` facts in each iteration, until no new facts are produced – a *fix-point* has been reached. This approach will inefficiently re-produce known facts in every iteration. We can instead use the well-known *Semi-Naïve Evaluation* (SN) [Ban86] which is efficient and produces no duplicates. Both the naïve evaluation and SN are *bottom-up* evaluation techniques, which start from the initial database and perform a repeated application of the rules until a fixpoint is reached.

Algorithm 1 is SN. M is the initial model (database), S contains all facts obtained thus far, δS and $\delta S'$ contain facts obtained during the previous and current iteration, respectively, and T_E and T_R are the Immediate Consequence Operator (ICO) for the exit rule(s) and the recursive rule(s), respectively. The algorithm evaluates as follows. Firstly, T_E (i.e. the exit rules) is applied to M to derive the first set of new δ facts δS (line 2). Then, until no new facts are derived during an iteration, T_R is evaluated on δS to derive new facts to be used in the next iteration. The new set of δ facts ($\delta S'$) is produced only after the removal of facts found in previous steps (line 5). Then, $\delta S'$ is merged into S (line 6) and becomes the set of facts used in the next iteration (line 7).

Next, we walk through an application of SN using Program 1 as our target. To enable SN,

[1] With set-containment monotonicity, evaluation only grows a predicate's set of facts.

Algorithm 1 Semi-Naïve Evaluation

1: $S := M$;
2: $\delta S := T_E(M)$;
3: $S := S \cup \delta S$;
4: **do**
5: $\quad \delta S' := T_R(\delta S) - S$;
6: $\quad S := S \cup \delta S'$;
7: $\quad \delta S := \delta S'$;
8: **while** ($\delta S \neq \varnothing$)
9: **return** S;

a (symbolic) rewriting [ZCF97] is applied to the rules of the original program to produce a new recursive rule that maintains program correctness. In the specific case of Program 1, the new rule only evaluates facts of the recursive predicate (tc) produced during the previous iteration (indicated with δ) and has the form $\texttt{tc}(\texttt{X}, \texttt{Y}) \leftarrow \delta\texttt{tc}(\texttt{X}, \texttt{Z}),\ \texttt{arc}(\texttt{Z}, \texttt{Y})$.

Algorithm 2 Semi-Naïve Evaluation of Program 1

1: $\delta\texttt{tc} := \texttt{arc}(\texttt{X}, \texttt{Y})$
2: $\texttt{tc} := \delta\texttt{tc}$
3: **do**
4: $\quad \delta\texttt{tc}' := \pi_{\texttt{X},\texttt{Y}}(\delta\texttt{tc}(\texttt{X}, \texttt{Z}) \bowtie \texttt{arc}(\texttt{Z}, \texttt{Y})) - \texttt{tc}$
5: $\quad \texttt{tc} := \texttt{tc} \cup \delta\texttt{tc}'$
6: $\quad \delta\texttt{tc} := \delta\texttt{tc}'$
7: **while** ($\delta\texttt{tc} \neq \varnothing$)
8: **return** tc

Algorithm 2 is the SN framework applied to Program 1. Note that the rules have been converted to a relational operator form (lines 1,4). In Algorithm 2 tc is the set of all facts produced for the recursive predicate and δtc (δtc$'$) is the set of facts produced for tc during the previous (current) iteration. The exit rules are evaluated first. The facts of arc become the initial set of facts for

both δtc (line 1) and tc (line 2). Then, SN iterates until a fixpoint is reached (line 7). Each iteration begins by joining δtc with arc and projecting X, Y terms to produce candidate tc facts (line 4). These facts are then set-differenced with tc to eliminate duplicates and produce δtc$'$ (line 4), which is unioned into tc (line 5) and becomes δtc (line 6) for the next iteration.

CHAPTER 3

The Deductive Application Language System

In this dissertation we describe our contributions toward the next generation of Datalog systems demonstrated by the *Deductive Application Language* ($DeAL$) and the $DeAL$ System[1] ($DeALS$) we have developed at UCLA. Unlike other recent systems that target DATALOG at a specific domain [SGL13, EF10], $DeALS$ has been designed as a general system seeking to satisfy many needs. In fact, $DeALS$ builds off the lessons learned in the course of a long experience with logic-based data languages, and the LDL [CGK90] and LDL++ [AOT03] experiences in particular, and is based on this earlier Deductive Database technology [AOT03]. Specifically, $DeAL$ is a DATALOG language that supports recursion, non-monotonic aggregates, negation and several advanced language constructs. These include user-defined aggregates, which enabled important knowledge discovery applications [GMT04], XY-stratified DATALOG [ZAO93], a form of local stratification that supports non-monotonic aggregation in recursion, choice [GZG92], which is backed by formal stable-model semantics and was found quite useful in program analysis [Hel10] and monotonic aggregates that can be used in recursive rules, which are the topic of Chapter 4.

In addition to a rich set of constructs, $DeALS$ was also designed to support a roster of optimization techniques including magic sets, supplementary magic sets and existential quantification. With support for many powerful constructs, a wide range of complex applications can be declaratively expressed and efficiently executed in our system. $DeAL$'s support for graph queries was demonstrated at VLDB [SZZ13].

[1] http://wis.cs.ucla.edu/deals

3.1 System Overview

In this section, after we briefly discuss the configurations for running $DeALS$, we review the type system and present a high level overview of the system architecture.

Using DeALS. $DeALS$ can be run as a standalone process and service requests from the $DeALS$ *driver*, which is an external API for $DeALS$. Using the driver, programmers can build client-server $DeALS$ applications, and in fact, this is how the $DeALS$ *command line client* is implemented. $DeALS$ can also be used as a library and embedded within an application. With this option, the $DeALS$ system API is programmed directly against and $DeALS$ runs within the application's process. $DeALS$ is implemented in Java and uses several third party libraries including ANTLR [ant15], Log4J [log15] and JUnit [jun15].

Type System. $DeALS$ supports a rich type system consisting of the following types: `byte`, {2, 4, 8, 16, 32} byte integers, `double` precision floating point decimal, `string`, `list`, `complex` (i.e., composite objects), and `datetime`. As a Datalog system, fast equality comparisons are essential and therefore variable-length types are encoded as a fixed-length value (e.g., as integer). For example, strings are variable-length and are dictionary encoded as integers. During compilation, data types for variables will be inferred starting from the definitions of the base relations up through the program rules.

Next, we discuss the three main components of the system – the compiler, the interpreter, and the storage manager.

Compiler. Given a database declaration, which consists of schema definitions for base relation(s), a program and a query form, the $DeAL$ compiler performs the following steps: First, the database declaration and program rules are parsed and the base relation definitions are loaded into the storage manager. An example database declaration for Program 1 is `database({arc(X : integer, Y : integer)})`. Second, rules with aggregates are expanded and rewritten into two rules, which is necessary to support the head aggregate syntax used by $DeAL$. An example of this *w.r.t.* monotonic aggregate rules can be found in Section 4.12. Next, the programs rules are analyzed and

grouped by predicate. Then, rules are formed into a Predicate Connection Graph (PCG), which is a type of AND/OR tree where OR nodes represent predicate occurrences in rule bodies and AND nodes represent rule heads [AOT03]. The PCG is analyzed and a binding passing analysis will be conducted to determine if any optimizations such as magic set rewriting (e.g., if the query form was presented with bound arguments) can be applied. Lastly, the Program Generator will use the PCG to produce an execution object graph of $DeAL$ interpreter framework objects, which we refer to as an *Interpreter Program* (IP). IP will be evaluated to produce the result set for the program.

Interpreter. The interpreter evaluates programs in main-memory and uses tuple-at-a-time pipelining. The interpreter is a bottom-up evaluator and evaluates nodes of IP in a left-to-right manner with intelligent backtracking [CGK90]. During evaluation, IP nodes retrieve and store facts in the storage manager.

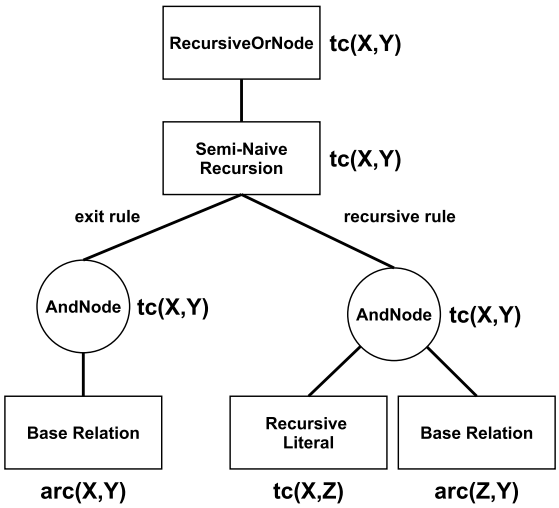


Figure 3.1: Interpreter Plan for Program 1.

Figure 3.1 is the IP for Program 1 and can be understood as follows. OR nodes are rectangles and AND nodes are circles. The compiler has identified the exit rule (*r1*) and recursive rule (*r2*) and placed their subtrees under the Semi-Naïve Recursion (OR) node. The exit rule subtree retrieves facts from the `arc` base relation and loads them into `tc`'s relation via the Semi-Naïve node. The recursive rule subtree representing *r2* will join the facts produced by the recursive literal and the `arc` base relation on `Z` and project facts with `(X,Y)` which will be added to `tc`'s relation.

Storage Manager. The $DeALS$ storage manager provides support for main memory storage and indexing for predicate relations. $DeALS$ supports several B+Tree data structure variants for tuple storage and indexing. A B+Tree stores fixed-size keys in internal and leaf nodes and non-key attributes in leaf nodes. Leaf nodes have pointers to their right neighbors to enable fast scanning of the tree. Through testing we determined our implementations perform best on average using a linear key search at both internal and leaf nodes with 256 bytes allocated for keys (e.g., 32 64-bit `long` keys) in each node, which results in shallow trees. $DeALS$ supports *B+Tree TupleStores*, which store tuples in a B+Tree. $DeALS$ also supports an *Unordered Heap TupleStore* (*UHT*) where tuples are stored as fixed-size entries in insertion order in large byte arrays. *UHT* can be given multiple indexes (e.g., B+Tree), which they remain synchronized with at all times. *UHT* enable a highwatermark approach for SN where each iteration is a contiguous range of tuples. Lastly, the interpreter uses nested loops joins and will automatically create an index and use an index nested loops join if the argument binding passing analysis identifies a bound join argument. For instance, for *r2* in Program 1, `arc` will be indexed on `Z` (first argument) which is bound by `tc`.

3.1.1 Declarative Optimizer

Although the Program Generator can be made to construct an optimized IP, our experience taught us that encoding the logic to optimize IP into the same procedure that construct IP can be difficult to implement and expensive to maintain. Continuing, rather than write the optimizer in Java, we instead added a declarative optimizer framework to $DeALS$. Using our framework, the logic to determine if an optimization should be applied is encoded as a $DeAL$ program. Procedural "hooks" are still needed, both to trigger the evaluation of the optimizer program and to apply the necessary changes to an IP based on the output of the optimizer program. Now, after the Program Generator produces an IP, the optimizer applies a set of optimizations to produce the final IP for the interpreter to evaluate.

To enable $DeAL$ programs to evaluate an IP, facts describing it are created and loaded into base relations. The schema for the optimizer base relations is displayed as Figure 3.2. Facts for base relations are produced by recursively traversing an IP.

```
predicate(predicateId:integer, name:string, depth:integer, type:string),
predicateDependency(predicateId:integer, childPredicateId:integer),
argument(argumentId:integer, type:string, name:string, datatype:string),
argumentPredicate(predicateId:integer, argumentId:integer),
binding(argumentId:integer, predicateId:integer, index:integer, binding:string)
```

Figure 3.2: Optimizer Schema.

Program 2 is an example $DeAL$ optimizer program to determine if a recursive predicate's node should be materialized instead of pipelined if there are multiple occurrences of the predicate in the program (i.e., the recursion will be computed multiple times otherwise).

Program 2. *$DeAL$ Optimizer Program to Materialize Recursion Nodes*

$$r1.\ \texttt{nodesToMaterialize}(\texttt{ID}) \leftarrow \texttt{predicate}(\texttt{ID}, \texttt{Name}, \texttt{D}, \text{`}\texttt{RecursiveOrNode}\text{'}),$$
$$\texttt{predicateCount}(\texttt{Name}, \text{`}\texttt{RecursiveOrNode}\text{'}, \texttt{N}), \texttt{N} > 1,$$
$$\texttt{predicateDependency}(\texttt{ID}, \texttt{ChildID}),$$
$$\texttt{predicate}(\texttt{ChildID}, \_, \texttt{D} + 1, \text{`}\texttt{SeminaiveNode}\text{'}),$$
$$\texttt{firstOccurrence}(\texttt{ID}, \texttt{Name}), \texttt{hasAllFreeBindings}(\texttt{ID}).$$

$$r2.\ \texttt{predicateCount}(\texttt{Name}, \texttt{Type}, \texttt{count}\langle\texttt{Name}\rangle) \leftarrow \texttt{predicate}(\_, \texttt{Name}, \_, \texttt{Type}).$$

$$r3.\ \texttt{firstOccurrence}(\texttt{ID}, \texttt{Name}) \leftarrow \texttt{predicate}(\texttt{ID}, \texttt{Name}, \_, \_),$$
$$\sim \texttt{earlierOccurrence}(\texttt{ID}, \texttt{Name}).$$

$$r4.\ \texttt{earlierOccurrence}(\texttt{ID}, \texttt{Name}) \leftarrow \texttt{predicate}(\texttt{ID2}, \texttt{Name}, \_, \_), \texttt{ID2} < \texttt{ID}.$$

$$r5.\ \texttt{hasAllFreeBindings}(\texttt{ID}) \leftarrow \texttt{allBindingCount}(\texttt{ID}, \texttt{N}), \texttt{freeBindingCount}(\texttt{ID}, \texttt{N}).$$

$$r6.\ \texttt{freeBindingCount}(\texttt{ID}, \texttt{count}\langle\texttt{ID}\rangle) \leftarrow \texttt{binding}(\_, \texttt{ID}, \_, \text{`}\texttt{f}\text{'}).$$

$$r7.\ \texttt{allBindingCount}(\texttt{ID}, \texttt{count}\langle\texttt{ID}\rangle) \leftarrow \texttt{binding}(\_, \texttt{ID}, \_, \_).$$

Program 2 is executed with the evaluation of the `nodesToMaterialize` predicate (*r1*). This predicate identifies a `RecursiveOrNode`, the node type that manages a recursive predicate, that has been added to the IP multiple times and has a child Semi-Naïve recursion node type that performs the actual recursion ($DeALS$ supports several types of recursive nodes). *r1* only produces the first occurrence of the node, since it will be evaluated first by the interpreter, and it must not have any bound arguments. Note, in *r3*, $\sim$ is the symbol used for negation in $DeAL$.

There are currently several optimizer programs and this list is expected to grow as new opti-

mization techniques are developed and as the system is extended to support new constructs. Examples of optimizer programs include:

- The counterpart of Program 2 - identify nodes to replace with read-only nodes that will read from the materialized node.
- Determining if a base relation is only used once and can have its secondary index removed and the relation converted to a more efficient B+tree storage structure.
- Identifying aggregate nodes to materialize because the same aggregate is used multiple times in the program.
- Identifying if a monotonic aggregate program (c.f. Chapter 4) that compares the produced aggregate value can be optimized for short-circuit evaluations.

CHAPTER 4

Optimizing Recursive Queries With Monotonic Aggregates in DeALS

4.1 Introduction

The growing demand for analytics has placed renewed focus on improving support for aggregation in recursion. Aggregates in recursive queries are essential in many important applications and are increasingly being applied in areas such as computer networking [LCG09] and social networks [SGL13]. Many significant applications require iterating over counts or probability computations, including machine learning algorithms for Markov chains and hidden Markov models, and data mining algorithms such as Apriori. Besides these new applications, we can mention a long list of traditional ones such as Bill of Materials (BOM), a.k.a. subparts explosion: this classical recursive query for DBMS requires aggregating the various parts in the part-subpart hierarchy. Finally, we have problems such as computing the shortest paths or counting the number of paths between vertices in a graph, which are now covered as foundations by most CS101 textbooks.

Although aggregates were not covered in E.F. Codd's definition of the relational calculi [Cod72], it did not take long before early versions of relational languages such as SQL included support for aggregate functions, namely `count`, `sum`, `avg`, `min` and `max`, along with associated constructs such as `group by`. However, a general extension of recursive query theory and implementation techniques to allow for aggregates proved an elusive goal, and even recent versions of SQL that provide strong support for OLAP and other advanced aggregates disallow the use of aggregates in recursion and only support queries that are stratified *w.r.t.* to aggregates.

The desirability of extending aggregates to recursive queries was widely recognized early and

many partial solutions were proposed over the years for Datalog languages [GZ01, Kol91, CM90, MPR90, MS95, RS92, ZCF97]. The fact that, in general, aggregates are non-monotonic w.r.t. set-containment led to proposals based on non-monotonic theories, such as locally stratified programs and perfect models [ZAO93, LLM98], well-founded models [Gel92] and stable models [FPL11]. An alternative approach was proposed by Ross and Sagiv [RS92], who observed that particular aggregates, such as continuous count, are monotonic in lattices other than set-containment and thus can be used in non-stratified programs. However practical difficulties with this approach were soon pointed out, namely that determining the correct lattices by programmers and compilers would be quite difficult [Van93], and this prevented their utilization in practical query languages for a long time. Fortunately, we recently witnessed some important developments change the situation. Firstly, Hellerstein et al., after announcing a resurgence of Datalog, showed that monotonicity in special lattices can be very useful in proving formal properties such as eventual consistency [CMA12]. Secondly, we see monotonic aggregates making a strong comeback in practical query languages thanks to the results published in [MSZ13a, MSZ13b] and in [SGL13], summarized next.

The formalization of monotonic aggregates proposed in [MSZ13a, MSZ13b] preserves monotonicity *w.r.t.* set-containment, and it is thus conducive to simplicity and performance that follow respectively from the facts that (i) users no longer have to deal with lattices, and (ii) the query optimization techniques, such as SN and magic sets remain applicable [MSZ13a]. SociaLite [SGL13] also made an important contribution by showing that shortest path queries and other algorithms using aggregates in recursion, can be implemented very efficiently so that in many situations using DATALOG becomes preferable to that of hand-coding Big Data analytics in some procedural language.

In this chapter, we describe how we introduced powerful monotonic aggregates and their accompanying efficient evaluation techniques into $DeALS$. We show how we retrofitted $DeALS$, which already supported a rich set of constructs and optimizations, to also support these new optimization techniques for monotonic aggregates. We demonstrate how $DeALS$ now achieves both performance and generality, and we will underscore this by comparing not only with SociaLite

but also with systems such as DLV[FPL08] and LogicBlox[GAK12] that realize different performance/generality tradeoffs.

Overview. The first of two main parts of this chapter begins with Section 4.2 which presents the syntax and semantics for the min (mmin) and max (mmax) monotonic aggregates. Section 4.3 discusses the evaluation and optimization of monotonic aggregate programs. Section 4.4 presents implementation details for mmin and mmax and the $DeALS$ storage manager. Section 4.5 presents experimental results for Sections 4.2-4.4. The second part of this chapter begins with Section 4.6 discussing the count (mcount) and sum (msum) monotonic aggregates, followed by their implementation in Section 4.7 and experimental validation in Section 4.8. Section 4.9 presents the formal semantics on which our aggregates are based. Section 4.10 describes how $DeAL$ programs are mapped into Datalog^{FS} programs. Section 4.11 presents additional optimizations for programs with monotonic aggregates. Section 4.12 presents rule rewriting techniques used to support monotonic aggregates. Section 4.13 provides additional $DeAL$ program examples. Additional related works are reviewed in Section 4.15 and we conclude in Section 4.16.

4.2 MMIN and MMAX Monotonic Aggregates

An mmin or mmax monotonic aggregate rule has the form:

$$\mathtt{p}(\mathtt{K}_1, \ldots, \mathtt{K}_m, \mathtt{aggr}\langle \mathtt{T} \rangle) \leftarrow \mathtt{Rule\ Body}.$$

In the rule head, $\mathtt{K}_1, \ldots, \mathtt{K}_m$ are the zero or more *group-by arguments* we also refer to as $\overline{\mathtt{K}}$, $\mathtt{aggr} \in \{\mathtt{mmax}, \mathtt{mmin}\}$ is the *monotonic aggregate*, and T, the *aggregate term*, is a variable.

The aggregate functions mmin and mmax map an input set or multiset, we will call G, to an output set, we will call D. Then, given G, for each element $g \in G$ mmin will put g into output set D if g is *less than the least value* mmin has previously computed (observed) for G. Similarly, given an input set G, for each element $g \in G$ mmax will put g in output set D if g is *greater than the greatest value* mmax has previously computed for G. The mmin and mmax aggregates are monotonic w.r.t. set-containment and can be used in recursive rules, and G should be viewed

as a set containing the union of all values for a single group (*group-by* key) *across* all iterations. These aggregates memorize the most recently computed value and thus require a single pass[1] over G. When viewed as a sequence, the values produced by `mmin` and `mmax` are monotonic.

4.2.1 Running Example

The All-Pairs Shortest Paths (APSP) program has received much attention in the literature [CM90, RS92, Gel92, GGZ91, SR91]. APSP calculates the length of the shortest path between each pair of connected vertices in a weighted directed graph.

Program 3. *APSP with* `mmin`

$$r1.\ \texttt{spaths}(\texttt{X}, \texttt{Y}, \texttt{mmin}\langle\texttt{D}\rangle) \leftarrow \texttt{edge}(\texttt{X}, \texttt{Y}, \texttt{D}).$$

$$r2.\ \texttt{spaths}(\texttt{X}, \texttt{Y}, \texttt{mmin}\langle\texttt{D}\rangle) \leftarrow \texttt{spaths}(\texttt{X}, \texttt{Z}, \texttt{D1}), \texttt{edge}(\texttt{Z}, \texttt{Y}, \texttt{D2}), \texttt{D} = \texttt{D1} + \texttt{D2}.$$

$$r3.\ \texttt{shortestpaths}(\texttt{X}, \texttt{Y}, \texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{spaths}(\texttt{X}, \texttt{Y}, \texttt{D}).$$

Program 3 is the $DeAL$ APSP program with the `mmin` aggregate. The `edge` predicate denotes the edges of the graph. The intuition for this program is as follows. In the recursion (*r1*, *r2*), an `spaths` fact will be derived if a path from `X` to `Y` is either i) new or ii) has length shorter than the currently known length from `X` to `Y`. *r1* finds the shortest path for each `edge`. *r2* is the left-linear recursive rule that computes new shortest paths for `spaths` by extending previously derived paths in `spaths` with an `edge`. Logically, this approach can result in many facts `spaths` for `X`, `Y`, each with a different length. Therefore, the program is stratified using a traditional (non-monotonic) `min` aggregate (*r3*) to select the shortest path for each `X`, `Y`.

APSP By Example. Next, we walk through an evaluation using SN for Program 3.

$$\texttt{edge}(\texttt{a}, \texttt{b}, 1).\ \ \texttt{edge}(\texttt{a}, \texttt{c}, 3).\ \ \texttt{edge}(\texttt{a}, \texttt{d}, 4).\ \ \texttt{edge}(\texttt{b}, \texttt{c}, 1).\ \ \texttt{edge}(\texttt{b}, \texttt{d}, 4).\ \ \texttt{edge}(\texttt{c}, \texttt{d}, 1).$$

Figure 4.1: `edge` Facts for Program 3.

First, *r1* in Program 3, the exit rule, is evaluated on the edge facts in Figure 4.1. In the rule head in *r1*, `X` and `Y`, the non-aggregate arguments, are the *group-by arguments*. The `mmin` aggregate is

[1] SQL 2003 max, min, count and sum aggregates on the `unlimited preceding` window are similar to $DeAL$'s monotonic aggregates.

applied to each of the six edge facts and six spaths facts are successfully derived (not displayed to conserve space) because no aggregate values had been previously computed (memorized) and each group (i.e. $(\mathtt{a}, \mathtt{b})$) was represented among the facts only once. For the spaths predicate, mmin is now initialized with a value for each group.

$$\mathtt{spaths}(\mathtt{a},\mathtt{c},2) \leftarrow \mathtt{spaths}(\mathtt{a},\mathtt{b},1), \mathtt{edge}(\mathtt{b},\mathtt{c},1), 2=1+1.$$

$$\mathtt{FAIL} \leftarrow \mathtt{spaths}(\mathtt{a},\mathtt{b},1), \mathtt{edge}(\mathtt{b},\mathtt{d},4), 5=1+4. \text{ [i]}$$

$$\mathtt{FAIL} \leftarrow \mathtt{spaths}(\mathtt{a},\mathtt{c},3), \mathtt{edge}(\mathtt{c},\mathtt{d},1), 4=3+1. \text{ [ii]}$$

$$\mathtt{spaths}(\mathtt{b},\mathtt{d},2) \leftarrow \mathtt{spaths}(\mathtt{b},\mathtt{c},1), \mathtt{edge}(\mathtt{c},\mathtt{d},1), 2=1+1.$$

Figure 4.2: Derivations of Program 3, *r2* - Iteration 1.

SN evaluates the recursive *r2* rule in Program 3 using the six spaths facts derived by *r1*. Figure 4.2 displays four derivations attempted by *r2* in its first iteration. Derivations not displayed failed to join spaths and edge facts. The first attempt results in a new spaths fact because $\mathtt{spaths}(\mathtt{a},\mathtt{c},2)$ has an aggregate value less than the previous value for $(\mathtt{a},\mathtt{c})$, which was 3 (from *r1*). The failures denoted [i] and [ii] occurred because the facts to be derived would have aggregate values not less than the previous value for $(\mathtt{a},\mathtt{d})$, which is 4. Finally, $\mathtt{spaths}(\mathtt{b},\mathtt{d},2)$ is derived ($2 < 4$ for $(\mathtt{b},\mathtt{d})$).

$$\mathtt{spaths}(\mathtt{a},\mathtt{d},3) \leftarrow \mathtt{spaths}(\mathtt{a},\mathtt{c},2), \mathtt{edge}(\mathtt{c},\mathtt{d},1), 3=2+1.$$

Figure 4.3: Derivations of Program 3, *r2* - Iteration 2.

Using the two facts derived in Figure 4.2, SN performs a second iteration using *r2*. As displayed in Figure 4.3, $\mathtt{spaths}(\mathtt{a},\mathtt{d},3)$ is derived because ($3 < 4$) for $(\mathtt{a},\mathtt{d})$. Now, no new facts can be derived and a fixpoint is reached.

$$\mathtt{shortestpaths}(\mathtt{a},\mathtt{c},2) \leftarrow \{\mathtt{spaths}(\mathtt{a},\mathtt{c},3), \mathtt{spaths}(\mathtt{a},\mathtt{c},2)\}$$

$$\mathtt{shortestpaths}(\mathtt{a},\mathtt{d},3) \leftarrow \{\mathtt{spaths}(\mathtt{a},\mathtt{d},4), \mathtt{spaths}(\mathtt{a},\mathtt{d},3)\}$$

$$\mathtt{shortestpaths}(\mathtt{b},\mathtt{d},2) \leftarrow \{\mathtt{spaths}(\mathtt{b},\mathtt{d},4), \mathtt{spaths}(\mathtt{b},\mathtt{d},2)\}$$

Figure 4.4: Derivations of Program 3, *r3*.

Lastly, *r3* is evaluated over the `spaths` facts derived during recursion and uses a stratified `min` aggregate to derive only the fact with the shortest path for each group. Figure 4.4 displays derivations of *r3* on groups that had multiple facts derived in recursion showing why rules like *r3* are necessary with our semantics. In Section 4.4, we will discuss optimizations so rules such as *r3* do not have to be evaluated.

4.3 Monotonic Aggregate Evaluation

In this section, we present optimized evaluation techniques for programs with monotonic aggregates.

SN (Chapter 2) can be used to evaluate $DeAL$ programs with monotonic aggregates. Symbolic differentiation rules [ZCF97] are applied to monotonic aggregate rules in a straightforward manner to produce rules for SN. However, even though SN efficiently evaluates general Datalog programs, monotonic aggregate programs can be evaluated with even greater efficiency than SN provides. The *max-based optimization* [MSZ13b] identified that counting only needs to be performed on maximum (max) values if only monotonic arithmetic and boolean functions are used. In this work, we expand this observation which we refer to as the *Monotonic Optimization*. The intuition behind the *Monotonic Optimization* is that with our monotonic aggregates, monotonicity is preserved and values other than the max (`mmax`) or min (`mmin`) will add no new results and thus can be ignored. Only the max (min) intermediate values need to be used in derivations to produce the final max (min) value. In fact, the last fact produced by the aggregate for a group contains the greatest (`mmax`) or least (`mmin`) aggregate value, making this fact the only fact for the group that we need to produce for the next iteration.

4.3.1 Monotonic Aggregate Semi-Naïve Evaluation

The Monotonic Optimization enables an optimized SN for monotonic aggregates we call *Monotonic Aggregate Semi-Naïve Evaluation* (MASN).

Algorithm 3 Monotonic Aggregate Semi-Naïve Evaluation (MASN)

1: $S := M$;
2: $\delta S := getLast(T_E(M))$;
3: $S := S \cup \delta S$;
4: **do**
5: $\quad \delta S' := getLast(T_R(\delta S)) - S$;
6: $\quad S := S \cup \delta S'$;
7: $\quad \delta S := \delta S'$;
8: **while** ($\delta S \neq \varnothing$)
9: **return** S;

Algorithm 3 is MASN, which closely resembles SN. The main difference from SN is that for MASN we use $getLast()$[2] to produce, from the input set, a set containing (i) all facts from predicates that do not have monotonic aggregates and (ii) the last derived fact for each group from monotonic aggregate predicates. Now after the T_E or T_R produces a set of facts, $getLast$ will be applied to produce the actual new $\delta S'$.

4.3.2 Eager Monotonic Aggregate Semi-naive Evaluation

MASN employs a level-by-level iteration boundary of a breadth-first search (BFS) algorithm where δ facts derived during the current iteration will be held for use until the next iteration. However, facts produced from monotonic aggregate rules can be used immediately upon derivation. Looking at the derivations in the walk-through evaluation of APSP in Section 4.2.1 one can see a case where SN, and in this case MASN as it would have evaluated the same as SN, did not capitalize on this property of monotonic aggregates.

Figure 4.5 shows the derivations of interest extracted from Figure 4.2. We see the second derivation performed using $\texttt{spaths}(\texttt{a}, \texttt{c}, 3)$ (from δS) and resulting in failure because the value for $(\texttt{a}, \texttt{d})$ was 4. However, at the time the derivation is attempted, $\texttt{spaths}(\texttt{a}, \texttt{c}, 2)$, the result of the

[2] $DeALS$ supports MASN by maintaining a single fact per group in $\delta S'$.

Program 3 *r2* evaluation with SN or MASN
spaths(a, c, 2) $\leftarrow$ spaths(a, b, 1), edge(b, c, 1), 2=1+1.
FAIL $\leftarrow$ **spaths(a, c, 3)**, edge(c, d, 1), 4=3+1.

Figure 4.5: Example of Iteration Boundary of MASN.

immediately previous derivation, existed. Had $\texttt{spaths}(\texttt{a}, \texttt{c}, 2)$ been used, $\texttt{spaths}(\texttt{a}, \texttt{d}, 3)$ would have been derived here, instead of requiring another iteration (Figure 4.3).

To further capitalize on the *Monotonic Optimization*, we propose *Eager Monotonic Aggregate Semi-Naïve Evaluation* (EMSN). With EMSN, facts produced from monotonic aggregate rules are immediately available to be used in derivations. EMSN evaluates recursive rules with monotonic aggregates in a fact-oriented (fact-at-a-time) manner and the facts to use in an iteration are determined by the set of groups (keys) that had aggregate facts derived during the previous iteration. With EMSN, derivations with monotonic aggregates are always performed with the current aggregate value for the group.

Algorithm 4 is EMSN. In EMSN, recursive rules with monotonic aggregates are evaluated fact-at-a-time and all other recursive rules are evaluated using SN. Rules are partitioned into two sets, each with its own ICOs – T_{E_A} and T_{R_A} are the ICO for the monotonic aggregate exit and recursive rules, respectively, and T_{E_N} and T_{R_N} are the ICO for the remaining exit and recursive rules, respectively. T_{R_A} will be applied on one fact at a time. δS_A and $\delta S'_A$ are the sets of facts obtained during the previous and current iteration, respectively, for the monotonic aggregate rules, while δS and $\delta S'$ are the sets of facts obtained during the previous and current iteration, respectively, for the remaining rules. $\delta S_{A_{Keys}}$ is the set of keys for the aggregate groups that had facts derived during the previous iteration. M is the initial model, S contains all facts obtained thus far. $newfact$ is a fact derived from a single application of T_{R_A}.

Important points of Figure 4 are as follows. We use $getKeys()$ to project out the aggregate value from the aggregate facts to produce the set of facts representing groups (keys) to use in derivations in the next iteration. For example, $getKeys(\{\texttt{spaths}(\texttt{a}, \texttt{b}, \texttt{1})\})$ would produce $\{\texttt{spaths}(\texttt{a}, \texttt{b})\}$. $getKeys()$ is applied to the set produced by $T_{E_A}(M)$ to produce the initial

Algorithm 4 Eager Monotonic Aggregate Semi-Naïve Evaluation (EMSN)

1: $S := M$;
2: $\delta S_A := T_{E_A}(M)$;
3: $\delta S_{A_{Keys}} := getKeys(\delta S_A)$;
4: $\delta S := T_{E_N}(M)$;
5: $S := S \cup \delta S \cup \delta S_A$;
6: **do**
7: **for all** $key \in \delta S_{A_{Keys}}$ **do**
8: **while** $(newfact := T_{R_A}(getFact(key)))$ **do**
9: $\delta S'_A := \delta S'_A \cup \{newfact\}$;
10: $\delta S' := T_{R_N}(\delta S) - S$;
11: $S := S \cup \delta S' \cup \delta S'_A$;
12: $\delta S_{A_{Keys}} := getKeys(\delta S'_A)$;
13: $\delta S := \delta S'$;
14: $\delta S'_A := \varnothing$;
15: **while** $(\delta S \neq \varnothing$ **and** $\delta S_{A_{Keys}} \neq \varnothing)$
16: **return** S;

$\delta S_{A_{Keys}}$ (line 3). T_{E_N} (remaining rules) is applied to M to produce the initial δS (line 4). Once in the recursion, individually, each key in $\delta S_{A_{Keys}}$ is used to retrieve its group's current fact from the aggregate relation ($getFact(key)$), which T_{R_A} is then applied to (line 8). Successful derivations result in $newfact$ being added to $\delta S'_A$ (line 9). Then, T_{R_N} (remaining rules) is applied to δS and duplicates are eliminated producing $\delta S'$, the set of non-monotonic-aggregate facts to be used in the next iteration (line 10). $getKeys(\delta S'_A)$ produces the set of keys to be used in derivations in the next iteration (line 12). This process repeats until no new facts are produced during an iteration.

Program 3 *r2* evaluation with EMSN
spaths(a, c, 2) ← `spaths(a, b, 1), edge(b, c, 1), 2 = 1 + 1.`
spaths(a, d, 3) ← **spaths(a, c, 2)**, `edge(c, d, 1), 3 = 2 + 1.`

Figure 4.6: EMSN Fact-at-a-time Efficiency.

Now, consider the same scenario from Figure 4.5, but this time using EMSN. As shown in Figure 4.6, now after $\texttt{spaths}(\texttt{a}, \texttt{c}, 2)$ is produced, it is immediately used in the next derivation resulting in $\texttt{spaths}(\texttt{a}, \texttt{d}, 3)$ being derived an iteration earlier than with SN or MASN. Moreover, $\texttt{spaths}(\texttt{a}, \texttt{c}, 3)$ will not be used in any further derivations as it would result in the derivations of facts that will not lead to a final answer now with the existence of $\texttt{spaths}(\texttt{a}, \texttt{c}, 2)$.

Discussion. With the application of the ICO for recursive monotonic aggregate rules (T_{R_A}) on an individual fact, rather than on a set of facts, EMSN can use facts immediately upon derivation. Although EMSN is based on SN, and therefore BFS, EMSN has depth-first search (DFS) characteristics. Like BFS, EMSN still uses a level-at-a-time (iteration) approach guided by facts in δS and δS_A that were derived during the previous iteration. However, because EMSN uses the most recent aggregate value for the group, regardless of when the value was computed, EMSN can evaluate deeper than a single level of the search space during an iteration of evaluation. The result is higher (`mmax`) or lower (`mmin`) aggregate values being derived earlier in evaluation, which in turn prunes the search space to avoid derivation of facts that will not result in final values.

We considered an alternative approach for EMSN that instead maintains the set of aggregate facts derived during an iteration ($\delta S'_A$) where modification to the aggregate relation results in either an update or insert to $\delta S'_A$. However, with each iteration $\delta S'_A$ would become δS_A and a new $\delta S'_A$ would be started, therefore every modification to the aggregate relation would require both δS_A and the new $\delta S'_A$ to be searched to be updated with the new value, even if the aggregate group is not present in δS_A. Although δS_A and $\delta S'_A$ are generally smaller than the aggregate relation, if an aggregate group has many new results during an iteration, efficiency gained from searching smaller sets instead of searching a larger aggregate relation to retrieve the aggregate value when needed (line 8 in Figure 4) would be offset by searching these sets many times. Furthermore, this requires a more complicated implementation to properly synchronize facts in multiple data structures.

4.4 MMIN and MMAX Implementation

This section contains details of our system implementation for supporting the `mmin` and `mmax` aggregates.

4.4.1 Storage Manager Extension

Early experimentation found aggregation using either i) *UHT* (Chapter 3) with B+Tree or Linear Hashing-based secondary indexes or ii) B+Tree TupleStores lacking in execution time performance. The *B+Tree Aggregator TupleStore* (*B+AT*) is a B+Tree TupleStore optimized for pipelined aggregation in recursive queries that provides both good read and write performance. *B+AT* store fixed-size keys in internal and leaf nodes and fixed-size aggregate values in leaf nodes. Keys are unique and only one aggregate value per key is maintained. Leaf nodes have pointers to their right neighbors and linear search is used in both internal and leaf nodes. In a *B+AT*, aggregation is performed in the leaves, therefore only one search of the tree is needed to retrieve the previous value, compare it with the new value and perform the update.

Unlike with *UHT*, facts in *B+AT* are not easily tracked by reference or range because of node splitting. Therefore, during evaluation of recursive queries with EMSN, after an aggregate value is modified, the *B+AT* inserts the modified entry's key into the set of keys, which is also a B+Tree[3], to process for the next iteration ($\delta S_{A_{Keys}}$ in Figure 4). This approach requires two tree searches with an aggregate value modification – one *B+AT* search which results in a modified aggregate value and one $\delta S_{A_{Keys}}$ search to record the key. Should no modification occur, then only the *B+AT* is searched. To retrieve aggregate facts to use in derivations (line 8 in Figure 4), a specialized cursor scans $\delta S_{A_{Keys}}$, using each key to retrieve the key's aggregate value from the *B+AT*.

Hash table approaches can be an appealing alternative to B+Trees. In Section 4.5 we present experimental results comparing $DeALS$ with a system that utilizes a hash table approach and highlight some of key differences between using B+Trees and hash tables.

[3]Since we scan the set, we use a B+Tree which stores the keys in order and can benefit EMSN.

4.4.2 `mmin` and `mmax` Implementation

The `mmin` and `mmax` implementation tracks the least (`mmin`) or greatest (`mmax`) value computed for each group where each group has one tuple in the TupleStore. We use a single relation schema with one column for each of the predicate's *group-by argument* and a column for the aggregate value. Specifically, *B+AT* keys are the group-by arguments with the aggregate value stored in the leaf. *UHT* are given indexes with the group-by arguments as keys. For instance, `spaths` in Program 3 uses *B+AT* with keys (`X`, `Y`) and each `X`, `Y` is stored with its current value (`D`) in a leaf node.

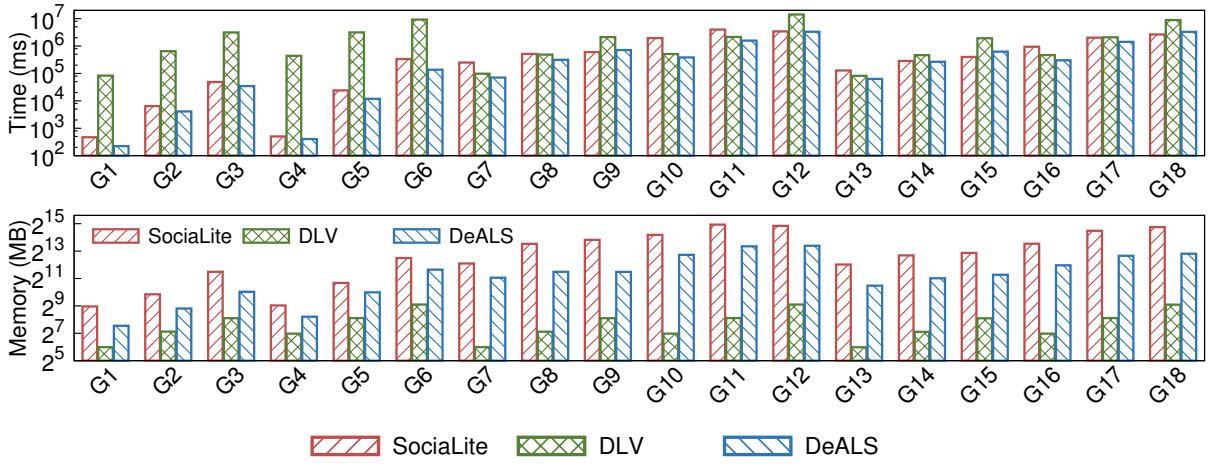
4.4.3 Operational Optimizations

Non-Redundant Relation Storage. Due to the *Monotonic Optimization*, we only need to maintain a single fact per group and when a new value for the group is successfully derived, we overwrite the previous value. If the recursive predicate and monotonic aggregate use separate stores, with EMSN and pipelining, the result is the recursive relation store is merely being synchronized with the aggregate relation store. Therefore, we do not allocate the recursive predicate a store, and instead have it read from the monotonic aggregate store.

Final Results via Monotonic Aggregate. Since the monotonic aggregate maintains the value for each group in its TupleStore, when a fixpoint is reached, its TupleStore contains the final results. For instance, instead of evaluating *r3* in Program 3 the recursion is materialized by the system, as it would have been by the stratified aggregate, and the final values are retrieved from the monotonic aggregate's TupleStore.

4.5 MMIN & MMAX Performance Analysis

All experiments on synthetic graphs were run on a machine with an i7-4770 CPU and 32GB memory running Ubuntu 14.04 LTS 64-bit. The experiments on real-life graphs were run on a machine with four AMD Opteron 6376 CPUs and 256GB memory running Ubuntu 12.04 LTS 64-



Directed acyclic graphs (DAGs): G1(10K/20K) G2(10K/50K) G3(10K/100K) G4(20K/40K) G5(20K/100K) G6(20K/200K)

Random graphs: G7(10K/20K) G8(10K/50K) G9(10K/100K) G10(20K/40K) G11(20K/100K) G12(20K/200K)

Scale-free graphs: G13(10K/20K) G14(10K/50K) G15(10K/100K) G16(20K/40K) G17(20K/100K) G18(20K/200K)

Figure 4.7: Execution time and memory utilization of APSP on synthetic graphs.

bit. Memory utilization is collected by the Linux `time` command. Execution time and memory utilization are calculated by performing the same experiment five times, discarding the highest and lowest values, and taking the average of the remaining three values. All experiments on systems written in Java were run using Java 1.8.0 except for SociaLite (0.8.1) which did not support Java 1.8.0. Experiments for SociaLite were run using Java 1.7.0.

Datasets. An n-vertex graph used in experiments has integer vertex labels ranging from 0 to $n-1$. We used three kinds of synthetic graphs — 1) directed acyclic graphs (DAGs), generated by connecting each pair of vertices i and j ($i < j$) with (edge) probability p; 2) random graphs, generated by connecting each pair of vertices with (edge) probability p; 3) scale-free graphs, generated using GTgraph[4]. The graphs are *shuffled* after generation where one random permutation is applied to the vertex labels and another random permutation is applied to the edges. The real-life graphs are not shuffled but we relabeled graphs whose vertex labels are beyond the range of [0, $n-1$] while

[4]GTgraph, `http://www.cse.psu.edu/~madduri/software/GTgraph/`.

maintaining the original edge order. A text description such as "10K/20K" indicates the graph has 10,000 vertices and 20,000 edges.

Configuration. *B+AT* and B+Tree indexes for *UHT* were configured with 256 bytes allocated for keys in each node (internal and leaf). Other than experiments in Section 4.5.2, $DeALS$ used EMSN with *B+AT*.

4.5.1 Datalog Implementation Comparison

$DeALS$ is a sequential, interpreted, main memory Java implementation. We compare $DeALS$ execution time and memory utilization performance against three Datalog implementations supporting aggregates in recursion — 1) the DLV system[5], the state-of-the-art implementation of disjunctive logic programming; 2) the commercial LogicBlox system, which supports aggregates in recursion using *staged partial fixpoint* semantics[6]. From log files produced during execution of recursive queries with aggregates, we determined LogicBlox indeed uses an approach akin to SN, which uses only new facts found in the current iteration in derivations in the next iteration; 3) the SociaLite[7] graph query language, which is compiled into code-generated Java, efficiently evaluates single-source shortest paths (SSSP) queries using an approach with Dijkstra's algorithm-like efficiency [SGL13] and supports a left-linear recursive APSP which it evaluates using SN. We used SociaLite 0.8.1 as it had the best sequential execution time performance of SociaLite versions available to us.

APSP on Synthetic Graphs. Figure 4.7 shows the results of APSP on synthetic graphs with random integer edge costs between 1-100 executed with $DeALS$, SociaLite and DLV. We experimented with two versions of LogicBlox — 3.10.15 and 4.1.3. The former does not support aggregates in recursion and although we can express APSP in a stratified program, it only terminates on DAGs, and only G1(0.286s) and G2(18.328s) finish within 24 hours. The latter supports aggregates

[5]DLV with recursive aggregates support, `http://www.dbai.tuwien.ac.at/proj/dlv/dlvRecAggr/`.

[6]LogicBlox 4 migration guide, `https://download.logicblox.com/wp-content/uploads/2014/05/LB-MigrationGuide-40.pdf`.

[7]`https://sites.google.com/site/socialitelang/`

in recursion however, only G1(4.809s), G2(6.697m), G7(4.774h) and G13(3.977h) finish within 24 hours. We do not report LogicBlox results in Figure 4.7 nor for the remaining experiments.

Among the 18 graphs described in Figure 4.7, we found SociaLite has the fastest execution time on three graphs and $DeALS$ has the fastest execution time on the remaining 15 graphs. $DeALS$ is more than two times faster than SociaLite on sparse graphs where the average degree of each vertex is only two (e.g., G7, G10 and G16). This advantage decreases as the average degree increases from two to ten. The main reason for this change is due to the different designs used by $DeALS$ and SociaLite. SociaLite uses an array of hash tables with an initial capacity of around 1,000 entries to maintain the delta relations, whereas $DeALS$ uses a B+Tree. The initialization cost of a hash table is higher than that of a B+Tree, while the cost of accessing a hash table is lower than that of a B+Tree. For graphs with small average degree, the initialization cost may account for a large percentage of the execution time, thus $DeALS$ is faster than SociaLite. The impact of the initialization cost reduces as the average degree increases, and thus SociaLite is faster than $DeALS$ on denser graphs. However, this faster execution time comes at the expense of higher memory utilization. SociaLite uses more than two times the memory as $DeALS$ on all 18 graphs. Although the C-based DLV has significantly lower memory utilization than both Java-based $DeALS$ and SociaLite, DLV is extremely slow compared with both $DeALS$ and SociaLite on DAGs. These results suggest that $DeALS$ achieves the best execution time versus memory utilization trade-off on sparse graphs among the three compared systems.

APSP on Real-life Graphs. Table 4.1 shows the results of APSP on three real-life graphs from the Stanford Large Network Dataset Collection[8]. The provided graphs do not have edge costs, therefore we assigned unit cost to each edge. The results are similar to that of synthetic graphs — $DeALS$ executes fastest while DLV has the lowest memory utilization. These results suggest that on real-life workloads the B+Tree-based design (low initialization cost) adopted by $DeALS$ is more favorable than the hash table-based design used by SociaLite.

SSSP on Real-Life Graphs. Figure 4.8 shows the results of SSSP on five real-life graphs from the

[8]Stanford large network dataset, `http://snap.stanford.edu/data/index.html`.

Table 4.1: Execution time and memory utilization of APSP on real-life graphs

	HepTh			Gnutella			Slashdot		
	S1	S2	S3	S1	S2	S3	S1	S2	S3
Time(h)	0.98	17.72	0.39	13.31	4.69	0.49	>24.00	>24.00	2.72
Mem(GB)	12.76	0.99	7.19	41.57	0.48	23.46	>89.59	>1.06	64.70

S1, S2, S3 represent SociaLite, DLV and $DeALS$ respectively.

HepTh(28K/353K): High-energy physics theory citation network.

Gnutella(63K/148K): Gnutella peer-to-peer network.

Slashdot(82K/549K): Slashdot social network.

USA road networks datasets[9]. For each graph, we evaluate SSSP on ten randomly selected vertices, and we report the min / (geometric) average / max execution time in the form of error bars. Since execution time captured for DLV includes time for loading the graph, evaluating the query and outputting the result, and query evaluation only accounts for a small percentage of overall time observed, timing for DLV is less informative for this experiment and thus we only report results for SociaLite and $DeALS$. SociaLite generates a Java program that evaluates the query using the Dijkstra's algorithm. The generated code achieves more than one order of magnitude speedup comparing to LogicBlox [SGL13]. However, our interpreted $DeALS$ is faster than the code-generated SociaLite for SSSP on the road network graphs as shown in Figure 4.8. This result is not surprising in the sense that EMSN optimizes SN, and the Bellman-Ford algorithm (equivalent to SN) usually yields comparable performance with the Dijkstra's algorithm on large sparse graphs.

USA road networks:
NY(264K/734K)
E(3,598K/8,778K)
W(6,262K/15,248K)
CTR(14,082K/34,292K)
USA(23,947K/58,333K)

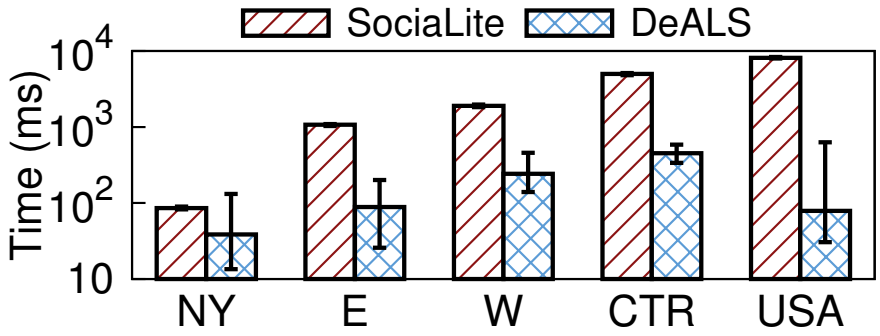


Figure 4.8: Execution time of SSSP on Road Networks Datasets.

[9]USA road networks, `http://www.dis.uniroma1.it/challenge9/download.shtml`.

4.5.2 $DeALS$ Storage Manager Evaluation

This experiment shows i) how EMSN performs relative to MASN and ii) how *B+AT* perform relative to *UHT*. We evaluated APSP on synthetic graphs G1, G2, G7, G8, G13 and G14 from Figure 4.7. The (geometric) average execution time and memory utilization over the six graphs are displayed in Table 4.2. Using *UHT* with B+Tree indexes, EMSN has a lower average execution time than MASN by 13%. A noticeable difference in performance is observed when using *B+AT* vs. *UHT* for EMSN. *B+AT* is 2.6 times faster than *UHT* and requires only approximately 25% of memory needed by *UHT*.

Table 4.2: Evaluation Type by Storage Configuration

Evaluation Type	**Storage Configuration**	**Time (s)**	**Memory (GB)**
MASN	*UHT* w/ B+Tree index	77.743	4.087
EMSN	*UHT* w/ B+Tree index	68.927	4.131
EMSN	*B+AT*	26.450	1.048

4.5.3 Statistical Analysis of Evaluation Methods

To provide a characterization of the relative performance of EMSN compared to SN, we perform an analysis over sets of graphs using the statistical estimation technique from [GKS91]. Statistics calculated[10] are 1) total number of facts derived by the recursive predicate (*derived facts*) and 2) total aggregate size of δS across all iterations (δ *facts*). These two statistics help to quantify the amount of computation the evaluation method must perform. For each statistic and each vertex number/edge probability combination, after each run of the program, the statistic is included in the average ($\bar{m}$) until a statistically significant number (30) of different graphs has been used AND $\bar{m}$ is within 5% error with 95% confidence[11].

[10] For these statistics, we assume a normal distribution ($N(\mu, \sigma^2)$, with mean μ and variance σ^2).

[11] As in [GKS91], $\bar{m}$ is accepted when $\epsilon < (0.05 * \bar{m})$, where $\epsilon = (1.96 * \sigma)/\sqrt{k}$. σ is standard deviation. k is the number of graphs used. 1.96, from the tables for standard normal distribution for 0.975, gives the 95% confidence coefficient.

We perform the analysis comparing APSP evaluated using EMSN to APSP evaluated using SN. We use randomly generated DAGs and random graphs with edge probability between 0.1 and 0.9 (increments of 0.1) and random integer edge cost between 1-50. EMSN and SN use the same sequence of graphs. On DAGs, SN requires 3-11% more *derived* and δ *facts* and on random graphs requires 13-18% more *derived* and δ *facts* than EMSN, respectively.

4.6 MCOUNT and MSUM Monotonic Aggregates

With efficient support for monotonic count and sum aggregates, $DeALS$ supports many exciting applications.

An `mcount` or `msum` monotonic aggregate rule has the form:

$$\mathtt{p}(\mathtt{K_1}, \ldots, \mathtt{K_m}, \mathtt{aggr}\langle(\mathtt{T}, \mathtt{P_T})\rangle) \leftarrow \mathtt{Rule\ Body}.$$

In the rule head, $\mathtt{K_1}, \ldots, \mathtt{K_m}$ are the zero or more *group-by arguments* ($\overline{\mathtt{K}}$), $\mathtt{aggr} \in \{\mathtt{mcount}, \mathtt{msum}\}$ is the *monotonic aggregate*, and $(\mathtt{T}, \mathtt{P_T})$ is the *aggregate term* pair passed from the body where T is a variable and $\mathtt{P_T}$ is a constant or a variable indicating the partial count/sum contributed by T.

As with `mmin` and `mmax`, the `mcount` and `msum` aggregates are monotonic w.r.t. set- containment and can be used in recursive rules. When viewed as a sequence, the values produced by `mcount` and `msum` are monotonic. The `mcount` and `msum` aggregate functions map an input set or multiset, we will call G, to an output set, we will call D. Elements $g \in G$ have the form (J, N_J), where N_J indicates the partial count/sum contributed by J. Note, J maps to T and N_J maps to $\mathtt{P_T}$ in the definition of the aggregate term above. Now, given G, for each element $g \in G$, if $N_J > N_{J_{prev}}$, where $N_{J_{prev}}$ is the memorized previous count (sum) for J or 0 if no previous count (sum) for J exists, `mcount` (`msum`) computes the count (sum) for G by summing the maximum partial count (sum) N_J for all J. Since only the maximum N_J for each J is summed, no double counting (summing) occurs. Lastly, `msum` only computes with positive numbers, thereby ensuring its monotonicity.

4.6.1 Running Example

Program 4 is the $DeAL$ program to count the paths between pairs of vertices in an acyclic graph. This program is not expressible with Datalog with stratified aggregation [MS95]. We will use Program 4 as our running example for `mcount` to explain monotonic counting in $DeAL$.

Program 4. *Counting Paths in a DAG*

$$r1.\ \texttt{cpaths}(\texttt{X},\texttt{Y},\texttt{mcount}\langle(\texttt{X},1)\rangle) \leftarrow \texttt{edge}(\texttt{X},\texttt{Y}).$$
$$r2.\ \texttt{cpaths}(\texttt{X},\texttt{Y},\texttt{mcount}\langle(\texttt{Z},\texttt{C})\rangle) \leftarrow \texttt{cpaths}(\texttt{X},\texttt{Z},\texttt{C}),\texttt{edge}(\texttt{Z},\texttt{Y}).$$
$$r3.\ \texttt{countpaths}(\texttt{X},\texttt{Y},\texttt{max}\langle\texttt{C}\rangle) \leftarrow \texttt{cpaths}(\texttt{X},\texttt{Y},\texttt{C}).$$

In Program 4, *r1* counts each edge as one path between its vertices. In *r2*, any $\texttt{edge}(\texttt{Z},\texttt{Y})$ that extends from a computed path count $\texttt{cpath}(\texttt{X},\texttt{Z},\texttt{C})$ establishes there are `C` distinct paths from `X` to `Y` through `Z`. The $\texttt{mcount}\langle(\texttt{Z},\texttt{C})\rangle$ aggregate in the head sums the count of paths from `X` to `Y` through every `Z` to produce the count from `X` to `Y`. Lastly, *r3* indicates only the maximum count for each path `X`, `Y` in `cpaths` is desired. As explained in Section 4.4.3, *r3* does not have to be evaluated.

Counting Paths By Example. Next, we walk through an evaluation of Counting Paths in Program 4 using EMSN to further explain `mcount`; this explanation is easily generalizable to `msum`. The `edge` facts in Figure 4.9 are the example dataset.

First, *r1* in Program 4 is evaluated and results in the six `cpaths` derivations as shown in the

Facts
$\texttt{edge}(\texttt{a},\texttt{b}).$
$\texttt{edge}(\texttt{a},\texttt{c}).$
$\texttt{edge}(\texttt{a},\texttt{d}).$
$\texttt{edge}(\texttt{b},\texttt{c}).$
$\texttt{edge}(\texttt{b},\texttt{d}).$
$\texttt{edge}(\texttt{c},\texttt{d}).$

Figure 4.9: `edge` Facts.

r1 Successful Derivations	Partial Count
$\texttt{cpaths}(\texttt{a},\texttt{b},1) \leftarrow \texttt{edge}(\texttt{a},\texttt{b}).$	$(\texttt{a},1)$ for $(\texttt{a},\texttt{b})$
$\texttt{cpaths}(\texttt{a},\texttt{c},1) \leftarrow \texttt{edge}(\texttt{a},\texttt{c}).$	$(\texttt{a},1)$ for $(\texttt{a},\texttt{c})$
$\texttt{cpaths}(\texttt{a},\texttt{d},1) \leftarrow \texttt{edge}(\texttt{a},\texttt{d}).$	$(\texttt{a},1)$ for $(\texttt{a},\texttt{d})$*
$\texttt{cpaths}(\texttt{b},\texttt{c},1) \leftarrow \texttt{edge}(\texttt{b},\texttt{c}).$	$(\texttt{b},1)$ for $(\texttt{b},\texttt{c})$
$\texttt{cpaths}(\texttt{b},\texttt{d},1) \leftarrow \texttt{edge}(\texttt{b},\texttt{d}).$	$(\texttt{b},1)$ for $(\texttt{b},\texttt{d})$
$\texttt{cpaths}(\texttt{c},\texttt{d},1) \leftarrow \texttt{edge}(\texttt{c},\texttt{d}).$	$(\texttt{c},1)$ for $(\texttt{c},\texttt{d})$

Figure 4.10: Derivations of Program 4, *r1* evaluation.

Figure 4.10. Each cpaths fact has a count of 1 indicating one path between each pair of vertices connected by edge facts. Displayed in the right column of Figure 4.10 is the memorized partial count (recall (J, N_J)) for each group. For example, in the first derivation, J=a, N_J=1, $(\texttt{a}, 1)$ is memorized for group $(\texttt{a}, \texttt{b})$.

r2 Successful Derivations	Partial Count
$\texttt{cpaths}(\texttt{a}, \texttt{c}, 2) \leftarrow \texttt{cpaths}(\texttt{a}, \texttt{b}, 1), \texttt{edge}(\texttt{b}, \texttt{c}).$	$(\texttt{b}, 1)$ for $(\texttt{a}, \texttt{c})$
$\texttt{cpaths}(\texttt{a}, \texttt{d}, 2) \leftarrow \texttt{cpaths}(\texttt{a}, \texttt{b}, 1), \texttt{edge}(\texttt{b}, \texttt{d}).$	$(\texttt{b}, 1)$ for $(\texttt{a}, \texttt{d})$*
$\texttt{cpaths}(\texttt{a}, \texttt{d}, 4) \leftarrow \texttt{cpaths}(\texttt{a}, \texttt{c}, 2), \texttt{edge}(\texttt{c}, \texttt{d}).$	$(\texttt{c}, 2)$ for $(\texttt{a}, \texttt{d})$*
$\texttt{cpaths}(\texttt{b}, \texttt{d}, 2) \leftarrow \texttt{cpaths}(\texttt{b}, \texttt{c}, 1), \texttt{edge}(\texttt{c}, \texttt{d}).$	$(\texttt{c}, 1)$ for $(\texttt{b}, \texttt{d})$

Figure 4.11: Derivations of Program 4, *r2* - Iteration 1.

EMSN evaluates the recursive *r2* rule from Program 4 using the cpaths derived by *r1*. Figure 4.11 shows the successful derivations performed by *r2*. As each cpaths fact is derived, it replaces the previous fact for the group (i.e. $\texttt{cpaths}(\texttt{a}, \texttt{c}, 2)$ replaces $\texttt{cpaths}(\texttt{a}, \texttt{c}, 1)$). Note the derivation of $\texttt{cpaths}(\texttt{a}, \texttt{d}, 2)$ from joining $\texttt{cpaths}(\texttt{a}, \texttt{b}, 1)$ and $\texttt{edge}(\texttt{b}, \texttt{d})$. It represents a count of two for $(\texttt{a}, \texttt{d})$, even though the rule body contributed only one path count. However, looking at the *-ed entry in Figure 4.10, we see a partial count of $(\texttt{a}, 1)$ towards $(\texttt{a}, \texttt{d})$ was accrued during evaluation of *r1*. Therefore, when computing the new count for $(\texttt{a}, \texttt{d})$, $(\texttt{a}, 1)$ and the newly derived $(\texttt{b}, 1)$ are summed to result in $\texttt{cpaths}(\texttt{a}, \texttt{d}, 2)$. Next, we observe the benefits of using EMSN with the derivation of $\texttt{cpaths}(\texttt{a}, \texttt{d}, 4)$. Since $\texttt{cpaths}(\texttt{a}, \texttt{c}, 2)$ existed even though it was derived this iteration, it was used and successfully joined with $\texttt{edge}(\texttt{c}, \texttt{d})$. Then, the partial counts for $(\texttt{a}, \texttt{d})$, which are $(\texttt{a}, 1)$, $(\texttt{b}, 1)$, and $(\texttt{c}, 2)$, are summed to produce $\texttt{cpaths}(\texttt{a}, \texttt{d}, 4)$. Finally, with no new facts produced after those in Figure 4.11, a fixpoint is reached, and since there is no need to evaluate *r3* (Section 4.4.3) we have our result.

4.7 MCOUNT and MSUM Implementation

In this section, we present implementation details for the `mcount` and `msum` aggregates. We use definitions from Section 4.6 (e.g., G). Note, G is a single group produced from the implicit *group-by* for a distinct assignment of $\overline{\texttt{K}}$, the zero or more *group-by* arguments. We will also refer to the TupleStore descriptions from Section 4.4.1. Lastly, although we use `mcount` to present our efficient count/sum technique, this discussion is generalizable to `msum`.

For `mcount` and `msum`, we use an approach based on delta-maintenance (Δ-*Maintenance*) techniques. Recalling our explanation for `mcount` in Section 4.6, given a new partial count $N_J > N_{J_{prev}}$, `mcount` will sum all maximum partial counts to compute the new total count for G. However, rather than recompute the total count, we can instead use Δ-*Maintenance* to increase N (the current total count for G) by $N_J - N_{J_{prev}}$ and put the updated count, now the total current count for G, into output set D. This produces the same result as if the maximum partial count N_J for all J are summed to produce the total count N for G, however avoids the re-summation of all N_J with each greater N_J. This requires memorizing both N for G and N_J for all J.

4.7.1 Storage Designs

Table 4.3 displays storage designs we investigated for `mcount` and `msum`. Here we use `N` to indicate the current count/sum for the group-by arguments ($\overline{\texttt{K}}$). As in Section 4.6, each `T` contributes a partial count/sum $\texttt{P}_\texttt{T}$ towards a distinct assignment of $\overline{\texttt{K}}$ (group).

Table 4.3: Storage Design Schemas

Name	**Schema**	**Indexes**
Double	$(\overline{\texttt{K}}, \texttt{N})$ | $(\overline{\texttt{K}}, \texttt{T}, \texttt{P}_\texttt{T})$	$\overline{\texttt{K}}$ | $(\overline{\texttt{K}}, \texttt{T})$
List	$(\overline{\texttt{K}}, \texttt{N}, \mathit{List}[(\texttt{T}, \texttt{P}_\texttt{T})])$	$\overline{\texttt{K}}$
B+Tree	$(\overline{\texttt{K}}, \texttt{N}, \mathit{B{+}Tree}[(\texttt{T}, \texttt{P}_\texttt{T})])$	$\overline{\texttt{K}}$
Hashtable	$(\overline{\texttt{K}}, \texttt{N}, \mathit{Hashtable}[(\texttt{T}, \texttt{P}_\texttt{T})])$	$\overline{\texttt{K}}$

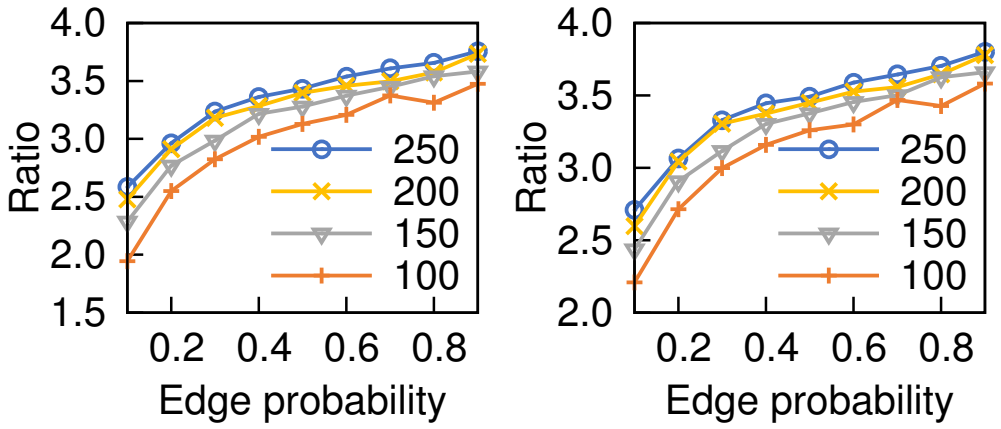
Double uses two relations, one relation $(\overline{\mathtt{K}}, \mathtt{N})$ indexed on $\overline{\mathtt{K}}$ to store tuples containing the group's total aggregate value and a second relation $(\overline{\mathtt{K}}, \mathtt{T}, \mathtt{P_T})$ indexed on $(\overline{\mathtt{K}}, \mathtt{T})$ to store the partial count $\mathtt{P_T}$ for each distinct assignment of $(\overline{\mathtt{K}}, \mathtt{T})$. Early testing showed *Double* using *UHT* without Δ-*Maintenance* taking 2-5 times longer to execute than with Δ-*Maintenance*.

We investigated designs using *KeyValue* type columns as a more efficient way of managing $(\mathtt{T}, \mathtt{P_T})$ pairs. We developed three single relation designs $(\overline{\mathtt{K}}, \mathtt{N}, KeyValue[(\mathtt{T}, \mathtt{P_T})])$, where N is the total count for $\overline{\mathtt{K}}$ and $KeyValue[(\mathtt{T}, \mathtt{P_T})]$ is a reference to the tuple's own *KeyValue*-type data structure. The relation is indexed on $\overline{\mathtt{K}}$ and each group has a single tuple. The *KeyValue*-types each represent a different retrieval time complexity; a *List* ($O(n)$) type, a *B+Tree* ($O(log(n))$) type, and a *Hashtable*($O(1)$) type. *Hashtable* is based on Linear Hashing and stores the hashed key in the bucket to avoid rehashing. *B+Tree* stores keys (T) in internal and leaf nodes and non-key attributes ($\mathtt{P_T}$) in leaf nodes, and uses linear search. Lastly, *List* stores $(\mathtt{T}, \mathtt{P_T})$ pairs ordered by T and uses a linear search. These are main memory structures, so designs attempt to limit the number of objects (e.g., *List* uses byte arrays). *KeyValue* designs use Δ-*Maintenance*.

For the designs shown in Table 4.3, $DeALS$ supports *List*, *HashTable* and *B+Tree* with *B+AT* and all designs with *UHT* indexed as shown. For example, using *r1*, *r2* from Program 4, with *B+Tree*, the *B+AT* would have $\mathtt{X}, \mathtt{Y}$ as keys and each entry in a leaf would have the current total count N and a reference to a B+Tree *KeyValue*-type to store $(\mathtt{T}, \mathtt{P_T})$ pairs. This design is essentially a B+Tree of B+Trees.

4.8 MCOUNT and MSUM Performance Analysis

Configuration. B+Tree TupleStores and indexes, *B+AT* and the *B+Tree KeyValue* design were configured with 256 bytes allocated for keys in each node (internal and leaf). The *Hashtable KeyValue* design used a directory and segment size of 256, 16 initial buckets and load factor of 10.



(a) Derived Facts.

(b) δ Facts.

Figure 4.12: Ratio SN/EMSN Derivations - Counting Paths.

4.8.1 Statistical Analysis of Evaluation Methods

We also perform the statistical analysis described in Section 4.5.3 comparing the evaluation of Counting Paths (Program 4) using EMSN to Counting Paths using SN. The experiment uses randomly generated DAGs of 100-250 vertices (increments of 50) and edge probability between 0.1 and 0.9 (increments of 0.1). EMSN and SN use the same sequence of graphs.

Figure 4.12a and Figure 4.12b show the results of the analysis. Each point on a line represents the ratio of SN to EMSN for number of *derived facts* (Figure 4.12a) or number of δ *facts* (Figure 4.12b) for the size of the graph indicated by the line and edge probability indicated by the x-axis. For example, in Figure 4.12b, SN produces more than three times as many δ facts as EMSN for graphs of 200 and 250 vertices starting at 0.2 (20%) edge probability. Figure 4.12a and Figure 4.12b show that for the test graphs, Counting Paths using SN derives 1.94 to 3.48 times more facts than EMSN. As edge probability increases, so does the ratio between SN and EMSN because the higher edge probability allows EMSN to derive and use facts earlier, which prunes the search space faster.

4.8.2 Storage Design Evaluation

This experiment tests how each of the storage designs presented in Section 4.7.1 performed on DAGs. Figure 4.13 shows the (geometric) average execution time and memory utilization, along with minimum and maximum values, on 45 random 250-vertex DAGs (5 graphs for each edge probability from 0.1 to 0.9) for each design. In Figure 4.13 results are shown in left-to-right order from worst to best average execution time performance.

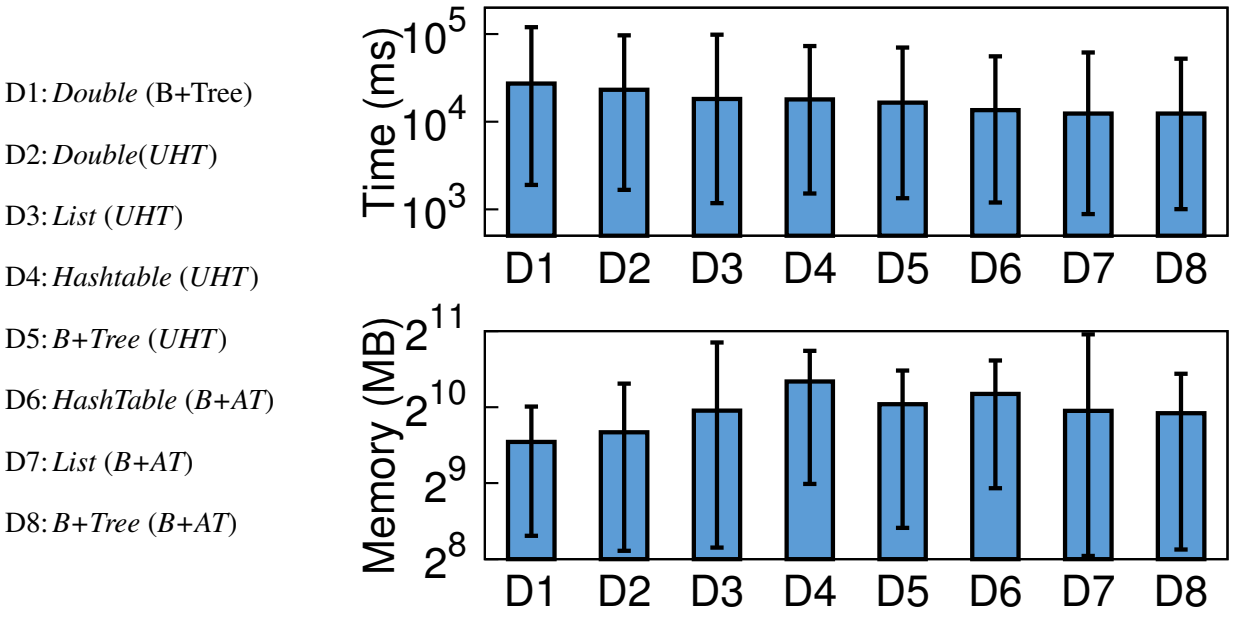


Figure 4.13: `mcount` and `msum` Storage Design Performance.

Recall the TupleStore descriptions from Section 4.4.1 and storage designs from Section 4.7.1. D2 is the *Double* design as described in Table 4.3 executed using *UHT* with B+Tree indexes. D1 is *Double* using a B+Tree TupleStore for the $(\overline{\mathtt{K}}, \mathtt{T}, \mathtt{P_T})$ relation[12]. D3-D5 and D6-D8 are *KeyValue* designs executed using *UHT* and *B+AT*, respectively. In Figure 4.13 we see D6-D8, the three *B+AT KeyValue* designs, have the best execution time performance. D7 and D8 have the lowest average execution time performance with *B+Tree* having better maximum (51s vs. 62s) execution time. Compared with D7, D8 has slightly better memory average utilization (969MB vs. 989MB) but lower maximum memory utilization (1.4GB vs. 2GB). Note, D1 and D2 have lowest average

[12]In *Double*, the $(\overline{\mathtt{K}}, \mathtt{T}, \mathtt{P_T})$ relation will contain many times more tuples than the $(\overline{\mathtt{K}}, \mathtt{N})$ relation (still *UHT*), so we focused on optimizing the larger.

memory utilization but their average execution times are nearly twice that of D7 and D8. Of the designs, D8, the *B+Tree* design using *B+AT*, best balances good average execution time performance with good average memory utilization.

4.8.3 Discussion

Finding other systems to perform an experimental comparison with `mcount` and `msum` proved challenging. Support in Datalog implementations for count and sum aggregates that can be used in recursion is not as mature as that of min and max aggregates. Using LogicBlox version 4, we were able to execute the Counting Paths program but experienced similar slow performance as with APSP (Section 4.5.1). Using SociaLite, we were unable to execute the Counting Paths program, and BOM queries such as subparts explosion, produced results different from ground truth. Lastly, we were able to execute a version of the Counting Paths program using DLV, but again the results were different from ground truth.

4.9 Formal Semantics

So far we have worked with the operational semantics of our monotonic aggregates and shown how this is conducive to the expression of algorithms by programmers. While most users only need to work at this level, it is important that we also show how this coincides with the formal semantics discussed in those two Datalog^{FS} papers [MSZ13a, MSZ13b], inasmuch as properties such as least fixpoint and stable models will follow from it.

We start with the example inspired by [RS92] (Party Invitations) for determining who will come to a party. In the $DeAL$ Program 5, some people will come to the party for sure, whereas others only join when at least three of their friends are coming. The idea is that with `cntComing` each person watches the number of their friends that are coming grow, and once that number reaches three, the person will then come to the party too. To count the number of friends, rather than the final count used in [RS92], we can use the `mcount` continuous count aggregate that enumerates all the integers until the actual maximum, i.e. it returns I, the actual maximum, representing the

integer interval $[1, I]$.

Program 5. *Who will come to the party?*

$r1.\texttt{willcome(X)} \leftarrow \texttt{sure(X)}.$

$r2.\texttt{willcome(X)} \leftarrow \texttt{cntComing(X,N)},\ \texttt{N} \geq 3.$

$r3.\texttt{cntComing(Y,mcount}\langle\texttt{X}\rangle) \leftarrow \texttt{friend(Y,X)},\ \texttt{willcome(X)}.$

Here the use of `mcount` over `count` is justified on the grounds of performance, since it is inefficient to count all friends of people if only three are required. More importantly though, while `count` is non-monotonic (unless we use the special lattices suggested by [RS92]), `mcount` is monotonic in the lattice of set containment used by the standard Datalog. So no ad hoc semantic extension is needed and concepts and techniques such as magic sets, perfect models and stable models can be immediately generalized to programs with `mcount`.

4.9.1 *DeAL* Interval Semantics

The lessons learned with `mcount` tell us that we can derive the monotonic counterpart of an aggregate by simply assuming that it produces an interval of integer values, rather than just one value. In the following we (i) apply this idea to max and min to obtain `mmax` and `mmin`, and then (ii) generalize these monotonic aggregates to arbitrary numbers, and show that the least fixpoint computation under this formal interval-based semantics can be implemented using the SN semantics used in Section 4.2 under general conditions that hold for all our examples. Due to space limitations the discussion is kept at an informal level: formal proofs are given in [MSZ13a, MSZ13b].

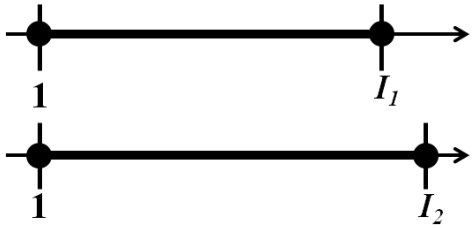


Figure 4.14: Counting Interval Semantics.

Before we introduce *DeAL*'s interval semantics, consider the following example of interval semantics for counting, based on the Datalog^{FS} interval semantics, depicted in Figure 4.14. If

r2 in Program 4 produced $\texttt{cpaths}(\texttt{a}, \texttt{b}, 3)$ and then $\texttt{cpaths}(\texttt{a}, \texttt{b}, 4)$, we cannot sum the aggregate values to get a new count for group $(\texttt{a}, \texttt{b})$. Instead, with the counts for $\texttt{cpaths}(\texttt{a}, \texttt{b}, 3)$ and $\texttt{cpaths}(\texttt{a}, \texttt{b}, 4)$ represented by $[1, 3]$ and $[1, 4]$, respectively, $[1, 3] \cup [1, 4] = max(3, 4) = 4$. Thus $\texttt{cpaths}(\texttt{a}, \texttt{b}, 4)$ represents $(\texttt{a}, \texttt{b})$'s count.

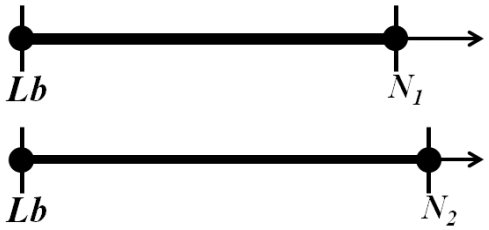


Figure 4.15: +/- Rational Numbers Interval Semantics.

+/- Rational Numbers. Computations on numbers in mantissa * exponent representation is *tantamount to integer arithmetic* on their numerators over a fixed large denominator. For instance, for floating point representations, the smallest value of exponent supported could be -95, whereby every number can be viewed as the integer numerator of a rational number with denominator 10^{95}. However, because of the limited length of the mantissa, this numerator will often be rounded-off — a monotonic operation that preserves the existence of a least-fixpoint semantics for our programs [MSZ13a]. Thus floating-point-type representation in programming languages can be used without violating monotonicity [MSZ13a, MSZ13b].

Now, say Lb represents the lower bound for all negative numbers supported by the system architecture. We can use the interval $[Lb, N]$ to represent any number regardless of its sign. The result of unioning the sets representing these numbers is a set representing the max, independent of whether this is positive or negative. Using Figure 4.15 as an example, with N_1 and N_2 represented by $[Lb, N_1]$ and $[Lb, N_2]$, respectively, then $[Lb, N_1] \cup [Lb, N_2]$ represents the larger of the two, i.e. $max(N_1, N_2) = N_2$. Thus, we can support negative numbers.

Minimum. Now, say Ub represents the upper bound for all positive numbers supported by the system architecture. We can represent the set of all numbers between N and Ub, i.e. the interval $[N, Ub]$, as the number N. Observe that a number smaller than N is represented by an interval that contains the interval $[N, Ub]$. As before, if we take a union of two or more such representations,

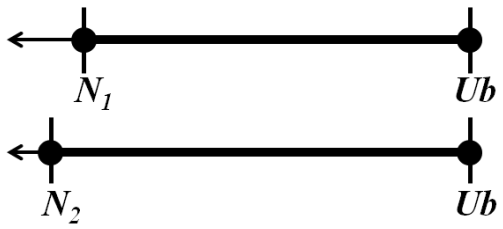


Figure 4.16: Minimum Interval Semantics.

the result is the *largest interval*. Using Figure 4.16 as an example, with N_1 and N_2 represented by $[N_1, Ub]$ and $[N_2, Ub]$, respectively, then $[N_1, Ub] \cup [N_2, Ub]$ represents the smaller of the two, i.e., $min(N_1, N_2) = N_2$.

As another example of these semantics, consider the first derivation in Figure 4.2. Here $\texttt{spaths}(\texttt{a}, \texttt{c}, 2)$ was derived because the previous value for $(\texttt{a}, \texttt{c})$ was 3. In the interval semantics, $\texttt{spaths}(\texttt{a}, \texttt{c}, 2)$ would be represented as $[2, Ub]$, and 3 as $[3, Ub]$, thus we have $[2, Ub] \cup [3, Ub]$ is $\texttt{min}(2, 3) = 2$.

4.9.2 Normal Programs

$DeAL$ programs that only use monotonic arithmetic and monotonic boolean (comparison) functions on values produced by monotonic aggregates will be called *normal*[13]. All practical algorithms we have considered only require the use of *normal* $DeAL$ programs. Two classes of *normal* programs exist.

Class 1. This class of *normal* programs uses monotonic aggregates to compute the max (min) values for use outside the recursion (e.g., to return the facts with the final max (min) values). Programs in this class include Programs 3 - 19 which are expressed using a stratified `max` (`min`) aggregate to select the max (min) value produced by the recursive rules when a fixpoint is reached. Examining the intermediate results of Class 1 programs: at each step of the fixpoint iteration, we have (i) the max (min) value and (ii) values less than the max value (greater than the min value). However, we do not need (ii), as long as the values in the head are computed from those in the

[13]The compiler can easily check if a program is normal when the program contains only arithmetic and simple functions (e.g. addition, multiplication).

body via an *arithmetic function that is monotonic*.

Class 2. Values produced by monotonic aggregates in Class 2 *normal* programs are not passed to rule heads, but are tested against conditions in the rule body. Here too, as long as the functions applied to the values are monotonic, rules are satisfied if and only if they are satisfied for the max (min) values. Program 5 is a Class 2 normal program.

4.9.3 Normal Program Evaluation

Recall the algorithm for MASN in Figure 3. Let us call L the set produced by $T_E(M)$ or $T_R(\delta S)$ and F the set produced from applying $getLast()$ to L. For SN, δS will be L, whereas for MASN, δS will be F. Let $W = L - F$. $W = \varnothing$ when, for facts of monotonic aggregate predicates, each group with facts derived during the iteration has only one fact. For an iteration, if $W = \varnothing$, MASN evaluates the program the same as SN. Otherwise, W contains facts that will not lead to final answers for Class 1 *normal* programs and for Class 2 *normal* programs, any condition satisfied (in a rule body) by a fact in W will also be satisfied by the fact of the same group in F. Thus, MASN does not need to evaluate W. In the next iteration, SN will derive all of the same facts as MASN, but also derive facts evaluating W. We have already established that these facts, because they were derived from W, will not lead to final answers, and thus MASN does not need to derive facts with these either.

Theorem 1. *A normal program with monotonic aggregates evaluated using MASN will produce the same result as that program evaluated using SN.*

Recall the algorithm for EMSN in Figure 4. Assume we are evaluating a *normal* program with SN. Let us call K_{SN} the set of facts used in derivations ($\cup$ of all δS) by SN. Now assume we are evaluating the same *normal* program with EMSN. Let us call K_{EMSN} the set of all facts used in derivations by EMSN, which means K_{EMSN} contains facts that were retrieved from the aggregate's relation, meaning they had the current value for the group at the time the fact was used in a derivation. Now, let $C = K_{SN} - K_{EMSN}$. If $C = \varnothing$, EMSN evaluates the program the same as SN. Otherwise, C contains facts that were not used by EMSN because at the time

of derivation, the values in the aggregate's relation for those facts' groups were greater (`mmax`, `mcount`, `msum`) or lesser (`mmin`) than the aggregate value in the fact. We know $K_{SN} \cap K_{EMSN} = K_{EMSN}$ and $K_{EMSN} \subset K_{SN}$ because EMSN will ignore facts to use in derivations that SN will use in derivations, but EMSN will not use different facts than SN. Stated another way, SN will attempt derivations with all facts EMSN attempts derivations with. Therefore, for Class 1 *normal* programs, facts in C are not going to lead to final answers. For Class 2 *normal* programs, any condition satisfied (in a rule body) by a fact in C would have also been satisfied by the value that was instead used. EMSN does not need to evaluate C. We have:

Theorem 2. *A normal program with monotonic aggregates evaluated using EMSN will produce the same result as that program evaluated using SN.*

4.10 Mapping $DeAL$ to Datalog^{FS}

In this section, we show how $DeAL$ programs can be mapped into Datalog^{FS} programs. Before presenting the transformation rules, we review Datalog^{FS} syntax and relevant language constructs.

4.10.1 Datalog^{FS}

Here we provide additional background on Datalog^{FS} syntax and constructs.

Syntax Like a Datalog program, a Datalog^{FS} program is a set of rules. Rule heads can either be an atom or an *FS-assert* statement. An *FS-assert* statement describes *multi-occurring predicates* and is an atom followed by : `K`, where `K` is either a constant or a variable. Body literals can be atoms, negated atoms or *FS Goals*. *FS Goals* can be either:

- *Running-FS Goal* with the form `K` : [`b-expression`]
- *Final-FS Goal* with the form `K` =![`b-expression`]

A *b-expression* is shorthand for *bracket expression*, which is either an individual or a conjunction of positive atoms. `K` :, the *Running-FS term*, is either a constant or a variable not contained

in the *b-expression*. This syntax allows for very concise program expression, especially when the program would otherwise require specifying a large number of conjunctions. For a *Running-FS Goal*, the head of the rule must belong to a stratum that is not lower than that of every predicate in the *b-expression*. For a *Final-FS Goal*, the head of the rule must belong to a stratum that is strictly higher than that of every predicate in the *b-expression*.

Running-FS Goals We review the *Running-FS Goal* construct using Program 6, the Datalog^{FS} version of the $DeAL$ program in Program 5 (also Program 8 from [MSZ13b]). This program finds friends who will go to a party if at least three of their friends will go. The predicate `friend(Y, X)` denotes person `Y` views person `X` as a friend. The predicate `sure(X)` denotes `X` will go to the party.

Program 6. *Who will come to the party? in Datalog^{FS}*

r1. `willcome(X)` ← `sure(X)`.

r2. `willcome(Y)` ← `3 : [friend(Y, X), willcome(X)]`.

In *r2*, the goal is a *Running-FS Goal*, with the *Running-FS term* of `3 :` and the *b-expression* of `friend(Y, X), willcome(X)`. The two kinds of variables in Datalog^{FS} are present in *r2*. `Y`, a *Global* variable, appears in the head of the rule (outside of the *b-expression*) and has rule-level scope. *Local* variable `X`, appears only within the *b-expression*, also its scope. In *r2*, with the *Running-FS term* of `3 :`, we view `X` as an existential variable enforcing the constraint that there must exist at least three distinct occurrences of `friend(Y, X), willcome(X)`.

To ensure the proper semantics of a program with a *Running-FS Goal*, *at least* `K` is the correct comparison. Because *Running-FS Goals* are monotonic and therefore, only an increasing number of new instances of distinct variable assignments will be found, using *equals* `K` or *at most* `K` could result in incorrect or unexpected program behavior. *Final-FS Goals* allow for the use of an equality comparison.

Final-FS Goals A *Final-FS Goal*, with the form `K =![b-expression]`, is used to find the total count of satisfying predicates from `b-expression`. This construct does not increase expressive power, but allows the programmer to express queries in a compact manner.

Program 7. *Counting Paths in a DAG in Datalog^{FS}*

$$r1.\texttt{path(X, Y)} : \texttt{1} \leftarrow \quad \texttt{edge(X, Y)}.$$
$$r2.\texttt{path(X, Y)} : \texttt{K} \leftarrow \quad \texttt{K} : [\texttt{edge(X, Z)}, \texttt{path(Z, Y)}].$$
$$r3.\texttt{countpaths(X, Y, K)} \leftarrow \texttt{K} = ![\texttt{path(X, Y)}].$$

In Program 7, the Datalog^{FS} version of the $DeAL$ program in Program 4, (also Program 17 in [MSZ13b]), $\texttt{K} = ![\texttt{path(X, Y)}]$ finds exactly the number K for each distinct variable assignments of X, Y that make $[\texttt{path(X, Y)}]$ true.

At this point, the reader has sufficient background on the syntax and language constructs of Datalog^{FS} so we can introduce the transformation rules to map $DeAL$ programs into Datalog^{FS} programs. Note, because Datalog^{FS} is based on monotonic *w.r.t.* set-containment semantics, we are only concerned with mapping $DeAL$ rules with `mmax` and `mcount` into Datalog^{FS} programs. We refer the reader to [MSZ13b] for more details on Datalog^{FS}.

4.10.2 Transformation Rules

Monotonic aggregates are found in three types of $DeAL$ program rules. If a monotonic aggregate is an argument in a rule head or an *MA-Value* is used in a rule goal, we say the head and/or the body is *FS*, otherwise, it is *Normal*. By typing $DeAL$ rules as such, we can identify which transformation rule to apply. Table 4.4 contains the rule types.

Rule Head	**Rule Body**	**Rule Type**
FS	Normal	Normal-to-FS
FS	FS	FS-to-FS
Normal	FS	FS-to-Normal

Table 4.4: Types of $DeAL$ Rules with Monotonic Aggregates

We have identified five transformation rules. Before we delve into the details of each rule, the following are some general rules of thumb. When transforming rule heads, monotonic aggregates are removed and corresponding *FS-assert* statements are added. When transforming rule bodies, first *MA-Values* are removed, then the goal is added to the *b-expression* of a *Running* or *Final-FS*

Goal. The variable removed from the goal will be assigned from the *FS goal*. Normal Datalog goals are left unchanged.

Our transformation rules are presented with rule bodies with singular goals. Cases for rule bodies with multiple goals can be easily extrapolated.

TR-N2FS.1 - ***Transformation Rule Normal-to-FS #1***

In rule type **Normal-to-FS** with an `mmax` aggregate in the $DeAL$ rule head, the `mmax` aggregates are removed and the head is made into an *FS-assert* statement.

$$\texttt{head}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}, \texttt{mmax}\langle\texttt{N}\rangle)$$

to

$$\texttt{head}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}) : \texttt{N}$$

TR-N2FS.2 - ***Transformation Rule Normal-to-FS #2***

Rule type **Normal-to-FS** with an `mcount` aggregate in the head requires removing the `mcount` aggregates from the head, making the head into an *FS-assert* statement, and adding a *Running-FS Goal* to the body to generate the *multi-occurring* predicate. The goals required to maintain the semantics of the original rule are added to the *Running-FS Goal*'s *b-expression*. The *Running-FS term* variable is the same as the variable in the *FS-assert* statement.

$$\texttt{head}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}, \texttt{mcount}\langle\texttt{X}\rangle)$$

where $\texttt{X}$ *is an argument* $\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}$ *from the body*

to

$$\texttt{head}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}) : \texttt{1}$$

However, if `X` is not a variable in the head, i.e. not a grouping variable, then body is treated as FS and **TR-FS2FS** is applied under the case for $\texttt{mcount}\langle(\texttt{Y}_\texttt{k}, \texttt{X})\rangle$. This ensures all eligible distinct assignments within the body will get counted and properly returned from the rule.

TR-FS2FS - *Transformation Rule FS-to-FS*

In rule type **FS-to-FS**, for the rule head, we do the same as in **TR-N2FS**, but the variable is renamed.

$$\texttt{head}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}, \texttt{mmax}\langle\texttt{X}\rangle)$$

or

$$\texttt{head}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}, \texttt{mcount}\langle(\texttt{Y}_\texttt{k}, \texttt{X})\rangle)$$

where $\texttt{Y}_\texttt{k}$ *is an argument from the body*

to

$$\texttt{head}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}) : \texttt{K}$$

For the rule body, for goals with an *MA-Value*, the argument is first removed from the goal, then the goal is added to the *b-expression* of the *Running-FS Goal*. Lastly, the variable from the *FS-assert* statement is set as the *Running-FS term*.

$$\texttt{goal}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}, \texttt{N})$$

where $\texttt{N}$ *is an MA-Value*

to

$$\texttt{K} : [\texttt{goal}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m})]$$

TR-FS2N.1 - *Transformation Rule FS-to-Normal #1*

In rule type **FS-to-Normal**, if the $DeAL$ rule head has a stratified `max` aggregate, it will be transformed into a Datalog^{FS} rule head as follows:

$$\texttt{head}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}, \texttt{max}\langle\texttt{N}\rangle)$$

to

$$\texttt{head}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}, \texttt{K})$$

where $\texttt{K}$ *is the result of a Final-FS Goal.*

The rule body will be transformed to have a *Final-FS Goal* with the goal added to the *b-expression* as follows:

However, if we used the standard $\texttt{goal}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}, \texttt{N})$

where N *is an MA-Value*

to

$\texttt{K=!}[\texttt{goal}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m})]$

TR-FS2N.2 - ***Transformation Rule FS-to-Normal #2***

A second case applies to **FS-to-Normal** rules if a goal's argument is an *MA-Value* which is evaluated inside the body and not passed to the head. Then, the head is left unchanged and the rule body will be transformed from:

$\texttt{goal}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m}, \texttt{N}), \texttt{N} \geq \texttt{C}$

where N *is an MA-Value and* C *is a constant*

to

$\texttt{C} : [\texttt{goal}(\texttt{Y}_1, \ldots, \texttt{Y}_\texttt{m})]$

4.10.3 Transformation Rules Examples

Using the transformation rules from Section 4.10.2, we now show step-by-step examples of $DeAL$ programs being transformed into Datalog^{FS} programs. The first example is Program 8, a Bill of Materials (BOM) example.

Program 8. *How many days until delivery?*

$r1.\texttt{delivery}(\texttt{Part}, \texttt{mmax}\langle\texttt{Days}\rangle) \leftarrow \texttt{basic}(\texttt{Part}, \texttt{Days}).$

$r2.\texttt{delivery}(\texttt{Part}, \texttt{mmax}\langle\texttt{Days}\rangle) \leftarrow \texttt{assbl}(\texttt{Part}, \texttt{Sub}, \_), \texttt{delivery}(\texttt{Sub}, \texttt{Days}).$

$r3.\texttt{actualDays}(\texttt{Part}, \texttt{max}\langle\texttt{Days}\rangle) \leftarrow \texttt{delivery}(\texttt{Part}, \texttt{Days}).$

r1 is a **Normal-to-FS** rule with an mmax aggregate therefore, **TR-N2FS.1** is applied. The monotonic aggregate in the head ($\texttt{mmax}\langle\texttt{Days}\rangle$) is removed and the rule head becomes an *FS-assert term* using the variable Days from the aggregate. r1 from Program 8 is converted from

$\texttt{delivery}(\texttt{Part}, \texttt{mmax}\langle\texttt{Days}\rangle) \leftarrow \texttt{basic}(\texttt{Part}, \texttt{Days}).$

to

$$\texttt{delivery}(\texttt{Part}) : \texttt{Days} \leftarrow \texttt{basic}(\texttt{Part}, \texttt{Days}).$$

Applying **TR-FS2FS** to *r2*, $\texttt{mmax}\langle\texttt{Days}\rangle$ is removed and Days is used in the *FS-assert* term. The $\texttt{delivery}(\texttt{Sub}, \texttt{Days})$ goal is added to the *b-expression* of a *Running-FS Goal*, but only after the Days argument is removed. Days is set as the *Running-FS term*. *r2* from Program 8 is converted from

$$\texttt{delivery}(\texttt{Part}, \texttt{mmax}\langle\texttt{Days}\rangle) \leftarrow \quad \texttt{assbl}(\texttt{Part}, \texttt{Sub}, \_), \texttt{delivery}(\texttt{Sub}, \texttt{Days}).$$

to

$$\texttt{delivery}(\texttt{Part}) : \texttt{Days} \leftarrow \quad \texttt{assbl}(\texttt{Part}, \texttt{Sub}, \_), \texttt{Days} : [\texttt{delivery}(\texttt{Part})].$$

Lastly, **TR-FS2N.1** is applied to *r3* because of the stratified max aggregate in the head. $\texttt{max}\langle\texttt{Days}\rangle$ is exchanged with Days in the head. Before the $\texttt{delivery}(\texttt{Part}, \texttt{Days})$ goal is added to the *b-expression* of a *Final-FS Goal*, Days is removed and is assigned the result of the *Final-FS Goal*.

$$\texttt{actualDays}(\texttt{Part}, \texttt{max}\langle\texttt{Days}\rangle) \leftarrow \texttt{delivery}(\texttt{Part}, \texttt{Days}).$$

to

$$\texttt{actualDays}(\texttt{Part}, \texttt{Days}) \leftarrow \texttt{Days} =![\texttt{delivery}(\texttt{Part})].$$

Program 9. *Program 8 transformed to Datalog^{FS}*

$r1.\ \texttt{delivery}(\texttt{Part}) : \texttt{Days} \leftarrow \quad \texttt{basic}(\texttt{Part}, \texttt{Days}).$

$r2.\ \texttt{delivery}(\texttt{Part}) : \texttt{Days} \leftarrow \quad \texttt{assbl}(\texttt{Part}, \texttt{Sub}, \_), \texttt{Days} : [\texttt{delivery}(\texttt{Part})].$

$r3.\ \texttt{actualDays}(\texttt{Part}, \texttt{Days}) \leftarrow \texttt{Days} =![\texttt{delivery}(\texttt{Part})].$

To show how **TR-FS2N.2** is applied, we walk through the transformation of the $DeAL$ Program 5, also displayed as Program 10 for the reader's convenience. **TR-FS2N.2** is needed for programs such as Program 10 that evaluate the result of an *FS Goal* inside the body, and do not pass it to the head, and therefore do not have an aggregate in the head requiring transformation.

Program 10. *Who will come to the party?*

$r1.\ \texttt{willcome}(\texttt{X}) \leftarrow \texttt{sure}(\texttt{X}).$

$r2.\ \texttt{willcome}(\texttt{X}) \leftarrow \texttt{cntComing}(\texttt{X}, \texttt{N}),\ \texttt{N} \geq 3.$

$r3.\ \texttt{cntComing}(\texttt{Y}, \texttt{mcount}\langle\texttt{X}\rangle) \leftarrow \texttt{friend}(\texttt{Y}, \texttt{X}),\ \texttt{willcome}(\texttt{X}).$

In Program 10, *r2* is an **FS-to-Normal** rule and **TR-FS2N.2** is applied. Since N in cntComing(Y, N) is an *MA-Value*, N is removed and cntComing(Y) is added to the *b-expression* of a *Running-FS Goal*. 3 is set as the *Running-FS term* and the comparison predicate is removed.

$$\texttt{willcome(Y)} \leftarrow \texttt{cntComing(Y, N)}, \texttt{N} \geq 3.$$

to

$$\texttt{willcome(Y)} \leftarrow 3 : [\texttt{cntComing(Y)}].$$

r3 is a **Normal-to-FS** rule, therefore **TR-N2FS** is applied. However, this is the special case where X is not in the head, therefore **TR-FS2FS** is applied. $\texttt{mcount}\langle\texttt{X}\rangle$ is removed from the head and an *FS-assert statement* is added with variable N. A *Running-FS Goal* is added to the body and since both goals are required to count the friends coming, both are added to the *b-expression*.

$$\texttt{countwillcome(Y, mcount}\langle\texttt{X}\rangle\texttt{)} \leftarrow \texttt{friend(Y, X)}, \texttt{willcome(X)}.$$

to

$$\texttt{countwillcome(Y)} : \texttt{N} \leftarrow \texttt{N} : [\texttt{friend(Y, X)}, \texttt{willcome(X)}].$$

After transforming *r2* and *r3*, we have an equivalent Datalog^{FS} program as shown in Program 11.

Program 11. *Program 5/10 transformed to Datalog^{FS}*

$$\begin{aligned}
&r1.\ \texttt{willcome(X)} \leftarrow && \texttt{sure(X)}.\\
&r2.\ \texttt{willcome(Y)} \leftarrow && 3 : [\texttt{cntComing(Y)}].\\
&r3.\ \texttt{cntComing(Y)} : \texttt{N} \leftarrow && \texttt{N} : [\texttt{friend(Y, X)}, \texttt{willcome(X)}].
\end{aligned}$$

Reducing Datalog^{FS} rules Although they are equivalent programs, the Program 11 has three rules, whereas the Program 6 has two rules. This is because the $DeAL$ program it originated from required an additional rule (r3) to compute the aggregation. While Datalog^{FS} rules can perform an aggregation, capture the result of the aggregation and perform a comparison against the result in one rule, such as *r2* in Program 6, $DeAL$'s head aggregate syntax doesn't allow for this. Instead, rules such as *r3* in Program 10 must be written to perform the aggregation, and a second rule, such as *r2* in Program 10, must receive the result and perform the comparison.

Fortunately, in Program 11 we can reduce the program to two rules without changing the meaning of the program. Because both rule bodies are a single *Running-FS Goal*, we substitute the *b-expression* of the *Running-FS Goal* in *r2* with the *b-expression* of the *Running-FS Goal* of *r3* and eliminate *r3*. Finally, the resulting program is the Datalog^{FS} Program 6.

This concludes our presentation of how to map $DeAL$ programs into Datalog^{FS} programs. Through syntactic transformation rules, we have shown how $DeAL$ programs are indeed equivalent to Datalog^{FS} programs. Combined with the semantics of $DeAL$'s monotonic aggregates from Section 4.9, the reader should recognize that $DeAL$ does indeed support Datalog^{FS}.

4.11 Additional Optimizations

In this section, we present optimizations for programs with monotonic aggregates.

4.11.1 Magic Sets

The preservation of the standard Datalog semantics via the interval semantics of our aggregates have made it possible for $DeALS$ to preserve the powerful optimization techniques of Datalog, one being the Magic Sets [BMS86] optimization. Furthermore, the use of head notation for monotonic aggregates simplifies the compile-time rewriting used to implement the Magic Set method. For example, with Program 8 to find out how long part p22 will require for delivery, we use the following query: ?actualDays(p22, AD). After a binding passing analysis, the compiler derives the following rules from the query form and *r2* respectively:

$$\texttt{m.delivery}(\texttt{p22}).$$
$$\textit{r2m}.\,\texttt{m.delivery}(\texttt{Sub}) \leftarrow \texttt{assbl}(\texttt{Part}, \texttt{Sub}), \texttt{m.delivery}(\texttt{Part}).$$

These Magic Set rules will generate all the subparts of p22, so that the final computation of *r3*, will now be filtered by an m.delivery predicate:

$$\textit{r1}.\,\texttt{delivery}(\texttt{Part}, \texttt{mmax}\langle\texttt{Days}\rangle) \leftarrow \texttt{basic}(\texttt{Part}, \texttt{Days}), \texttt{m.delivery}(\texttt{Part}).$$
$$\textit{r2}.\,\texttt{delivery}(\texttt{Part}, \texttt{mmax}\langle\texttt{Days}\rangle) \leftarrow \texttt{assbl}(\texttt{Part}, \texttt{Sub}, \_), \texttt{delivery}(\texttt{Sub}, \texttt{Days}), \texttt{m.delivery}(\texttt{Part}).$$

As an additional example, using Program 4 we can find the number of paths from node7 with the query ?countpaths(node7, Y, C). The compiler will generate the following:

$$\texttt{m.cpaths}(\texttt{node7}).$$
$$r3.\ \texttt{countpaths}(\texttt{node7}, \texttt{Y}, \texttt{max}\langle(\texttt{C})\rangle) \leftarrow \texttt{cpaths}(\texttt{node7}, \texttt{Y}, \texttt{C}).$$

The first argument of $r3$ is bound to node7 and the magic predicate containing node7 is generated and added to the body of both *r1* and *r2* as follows:

$$r1.\ \texttt{cpaths}(\texttt{X}, \texttt{Y}, \texttt{mcount}\langle\texttt{X}\rangle) \leftarrow \quad \texttt{m.cpaths}(\texttt{X}), \texttt{arc}(\texttt{X}, \texttt{Y}).$$
$$r2.\ \texttt{cpaths}(\texttt{X}, \texttt{Z}, \texttt{mcount}\langle(\texttt{Y}, \texttt{C})\rangle) \leftarrow \texttt{m.cpaths}(\texttt{X}), \texttt{cpaths}(\texttt{X}, \texttt{Y}, \texttt{C}), \texttt{arc}(\texttt{Y}, \texttt{Z}).$$

As the magic set contains only the constant node7, the original program can now be specialized as follows:

$$r1.\ \texttt{cpaths}(\texttt{node7}, \texttt{Y}, \texttt{mcount}\langle\texttt{X}\rangle) \leftarrow \quad \texttt{arc}(\texttt{node7}, \texttt{Y}).$$
$$r2.\ \texttt{cpaths}(\texttt{node7}, \texttt{Z}, \texttt{mcount}\langle(\texttt{Y}, \texttt{C})\rangle) \leftarrow \texttt{cpaths}(\texttt{node7}, \texttt{Y}, \texttt{C}), \texttt{arc}(\texttt{Y}, \texttt{Z}).$$

4.11.2 Comparison-Only Monotonic Aggregation

In cases where an *MA-Value* is only evaluated inside the body of a rule, and not passed to the head, there is an opportunity for optimization. For example, Program 12 is the hindex query to find all researchers with at least 13 publications that have been cited 13 times. The refer relation contains the citations from PnFro to PnTo and the author relation contains the author(s) of each paper Pno. *r1* counts the number of citations paper number PnTo has received from other papers. *r2*, counts the number of papers cited at least 13 times for each author. Finally, *r3* determines if an author has at least 13 papers cited 13 times.

Program 12. *Find researchers with hindex of at least 13*

$$r1.\ \texttt{numCite}(\texttt{PnTo}, \texttt{mcount}\langle\texttt{PnFro}\rangle) \leftarrow \texttt{refer}(\texttt{PnFro}, \texttt{PnTo}).$$
$$r2.\ \texttt{numCite13X}(\texttt{Auth}, \texttt{mcount}\langle\texttt{Pno}\rangle) \leftarrow \texttt{author}(\texttt{Auth}, \texttt{Pno}), \texttt{numCite}(\texttt{Pno}, \texttt{N}), \texttt{N} >= 13.$$
$$r3.\ \texttt{hindex13}(\texttt{Auth}) \leftarrow \texttt{numCite13X}(\texttt{Auth}, \texttt{K}), \texttt{K} >= 13.$$

Notice in *r3*, K is not passed to the head, but only evaluated inside the body (K $>=$ 13). After K reaches 13 for an assignment of Auth, it will stay at 13 or increase, and $>=$ is a monotonic

boolean function such that once it is true, it will stay true forever. These stable properties ensure that no further evaluation need be performed for this author. However, the aggregate is unaware of this as it is simply executing the underlying `author` predicate, which will assign `Auth` to the next author only after it has exhausted all `Pno` for the current `Auth`[14]. A change in the assignment of `Auth` must be forced, which can be achieved using rewriting as follows:

$$r3.\ \texttt{hindex13(Auth)} \leftarrow \texttt{hindex13\_author(Auth)}, \texttt{numCite13X(Auth, N)}, \texttt{N} >= 13.$$

$$r3c.\texttt{hindex13\_author(Auth)} \leftarrow \texttt{author(Auth, \_)}.$$

Now `hindex13_author`, which is the set of all authors, is responsible for assigning `Auth`. And, once the predicate `N` $>=$ `13` in *r3* is satisfied, the next execution of *r3* will result in an assignment of `Auth` to the next author. The execution trade off here is that although a scan of `author` is still being performed, now by `hindex13_author`, an index lookup is now used on `author` and the program only aggregates the minimum count necessary to satisfy the `N` $>=$ `13` condition.

We conducted an experiment of this approach using a citation network[15]. After a best-effort extraction of the authors from the abstracts, we computed hindexes up to 45 on the graph. We found this technique to have better performance up until hindex 9 ($< 1\%$ of authors had 9), at which point it became more costly in terms of query execution time by on average 10%.

4.12 Monotonic Aggregate Rule Rewriting

We utilize compiler rewriting techniques to provide support for monotonic aggregate functions in $DeALS$. Our rewriting approach relies on a combination of straightforward rule rewriting techniques paired with specialized built-in predicates. After rewriting monotonic aggregate rules, no changes are required to rules invoking monotonic aggregate rules because the original signature (predicate name and arity) is left unchanged by the rewriting. All rewriting is done during compilation before program execution begins.

[14] base relations are sorted

[15] http://snap.stanford.edu/data/cit-HepTh.html

4.12.1 Rewriting `mmax` Rules

After the compiler validates the aggregate rules, the compiler rewrites an aggregate rule into two rules. We will explain our approach by applying the rewriting to *r1* from Program 8, also below for the reader's convenience.

$$r1.\texttt{delivery}(\texttt{Part}, \texttt{mmax}\langle\texttt{Days}\rangle) \leftarrow \texttt{basic}(\texttt{Part}, \texttt{Days}).$$

From *r1*, the compiler produces the rules in Program 13.

Program 13. *Rules produced from rewriting r1*

$$r1_1.\texttt{fs\_aggr\_delivery}(\texttt{Part}, \texttt{Days}, \texttt{nil}) \leftarrow \texttt{basic}(\texttt{Part}, \texttt{Days}).$$
$$r1_2.\texttt{delivery}(\texttt{Part}, \texttt{AggrVal}) \leftarrow$$
$$g1.\ \texttt{fs\_aggr\_delivery}(\texttt{Part}, \texttt{Days}, \texttt{OldMaxVal}),$$
$$g2.\ \texttt{fsmax}(\texttt{Days}, \texttt{OldMaxVal}, \texttt{NewMaxVal}),$$
$$g3.\ \texttt{fs\_aggr\_delivery}(\texttt{Part}, \texttt{NewMaxVal}, \texttt{AggrVal}).$$

In Program 13, $r1_1$ is the *Inner rule*, which is given the body of the original rule. $r1_2$ is the *Outer rule*. The first goal of the Outer rule, $g1$, is a special built-in predicate[16], called a *Read-Aggregate* predicate. For a $\overline{\texttt{K}}$ (`Part`) this predicate will 1) compute the next value to aggregate by executing the Inner rule ($r1_1$) and 2) retrieve the previous value from the aggregate's storage, if a value exists. A *Read-Aggregate* predicate is true when a new value is found.

If $g1$ is true, `Days` will be bound to a new value and `OldMaxVal` will be bound to the previous value for the assignment of `Part` or to `nil` if there was no previous value. $g2$, the `fsmax` built-in predicate, compares `Days` and `OldMaxVal` to determine if `Days` is indeed a new maximum value. If `OldMaxVal` is `nil` or `Days` $>$ `OldMaxVal`, then $g2$ is true and `NewMaxVal` is bound to the value assigned `Days`, otherwise, $g2$ is false. $g3$ is also a special built-in predicate called a *Write-Aggregate* predicate, which writes the value assigned to `NewMaxVal` to the aggregate's storage as the value for the assignment of `Part`. This predicate will also bind `AggrVal` to the value assigned

[16] built-in predicates enable rapid prototyping of different logical and physical implementations. The rule rewriting processes are insulated from changes with each new approach - especially helpful for prototyping `mcount`.

to NewMaxVal[17]. Invocations of the *Outer rule* return true when a new maximum value is found for the assignment of Part.

4.12.2 Rewriting mmin Rules

The rewriting of aggregate rules with mmin is similar to that of rules with mmax. Looking at Program 13, for *g2*, in place of the mmax built-in predicate, the mmin built-in predicate with the same form will be used. The mmin built-in predicate compares its first and second arguments and will be true when the second argument is nil or when the first argument is less than the second argument. When true, the third argument will be bound to the value assigned to the first argument. Variables will be named accordingly (OldMinVal instead of OldMaxVal, NewMinVal instead of NewMaxVal).

4.12.3 Rewriting mcount Rules

Aggregate rules with mcount are also rewritten in a way similar to rules with mmax. The semantics of mcount require producing only the maximum value, therefore mcount rewritten rules also utilize the fsmax built-in predicate. The noticeable difference in the rules produced for mcount is that because the mcount aggregate term is a pair (Section 4.6), the pair is also written into the predicate arguments in the new rules.

Program 14. *Form of aggregate rule with* mcount

$$\mathtt{p}(\mathtt{K}_1, \ldots, \mathtt{K}_m, \mathtt{mcount}\langle(\mathtt{J}, \mathtt{N})\rangle) \quad \leftarrow \mathtt{Rule\ Body}.$$

Program 14 is the form of an aggregate rule with mcount. In the aggregate term $(\mathtt{J}, \mathtt{N})$, J is an extra grouping argument. mcount utilizes different built-in *Read-Aggregate* and *Write-Aggregate* predicates than mmax and mmin. The mcount *Read-Aggregate* predicate retrieves the value stored for the combined $\overline{\mathtt{K}}$ and J. The mcount *Write-Aggregate* predicate writes the new value for the combined $\overline{\mathtt{K}}$ and J to the aggregate's storage and then binds the AggrVal argument to the

[17] Using AggrVal and NewMaxVal instead of having NewMaxVal in the head keeps mmax and mcount rule rewriting consistent.

aggregated value for only the $\overline{\texttt{K}}$.

As a concrete example of `mcount` rewriting, consider the rules produced by rewriting *r2* from Program 4 (also below) in Program 15.

$$r2.\ \texttt{cpaths}(\texttt{X}, \texttt{Z}, \texttt{mcount}\langle(\texttt{Y}, \texttt{C})\rangle) \leftarrow \texttt{cpaths}(\texttt{X}, \texttt{Y}, \texttt{C}), \texttt{arc}(\texttt{Y}, \texttt{Z}).$$

Program 15. *Rewritten rules from r2 in Program 4*

$$r2_1.\ \texttt{fs\_aggr\_cpaths\_2}(\texttt{X}, \texttt{Z}, (\texttt{Y}, \texttt{C}), \texttt{nil}) \leftarrow \texttt{cpaths}(\texttt{X}, \texttt{Y}, \texttt{C}), \texttt{arc}(\texttt{Y}, \texttt{Z}).$$

$$r2_2.\ \texttt{cpaths}(\texttt{X}, \texttt{Z}, \texttt{AggrValue}) \leftarrow$$

$$g1.\ \texttt{fs\_aggr\_cpaths\_2}(\texttt{X}, \texttt{Z}, (\texttt{Y}, \texttt{C}), \texttt{OldMaxVal}),$$

$$g2.\ \texttt{fsmax}(\texttt{C}, \texttt{OldMaxVal}, \texttt{NewMaxVal}),$$

$$g3.\ \texttt{fs\_aggr\_cpaths\_2}(\texttt{X}, \texttt{Z}, (\texttt{Y}, \texttt{NewMaxVal}), \texttt{AggrVal}).$$

There are several things to take note of in Program 15. First, the original rule *r2* was a recursive rule, so $r2_1$ and $r2_2$ are mutually recursive. Next, the rewriting has separated *r2* into an aggregate relation (`fs_aggr_cpaths_2`) and a recursive relation (`cpaths`). Third, the complete aggregate term $(\texttt{Y}, \texttt{C})$ was copied to the head of the Inner rule and to $g1$, the *Read-Aggregate* predicate, which will assign these variables. The term is kept as a pair so the compiler can more easily identify it as a value instead of as a key variable. Lastly, because *r2* is the second rule in its relation, the rule head for the Inner rule $r2_1$, as well as the goals for the *Read-Aggregate* ($g1$) and *Write-Aggregate* ($g3$) predicates, have had a `_2` suffix added to ensure that Outer rules only call Inner rules created from rewriting the same original rule.

4.12.4 Rewriting `msum` Rules

Aggregate rules with `msum` are rewritten similarly to `mcount` rules. As reviewed in Section 4.6, `msum` is operationally the same as `mcount` with the exception that `msum` operates on positive numbers only. Therefore, for rules with `msum`, a predicate is added to the body to ensure only positive numbers are being aggregated, thereby maintaining `msum`'s monotonicity.

4.13 Additional $DeAL$ Programs

This section includes additional programs to show $DeAL$'s expressiveness and support for a variety of applications.

Program 16. *Company Control*

$$r1.\ \texttt{cshares}(\texttt{A},\texttt{B},\texttt{dirct},\texttt{mmax}\langle\texttt{P}\rangle) \leftarrow \texttt{ownedshares}(\texttt{A},\texttt{B},\texttt{P}).$$

$$r2.\ \texttt{cshares}(\texttt{A},\texttt{C},\texttt{indrct},\texttt{mcount}\langle(\texttt{B},\texttt{P})\rangle) \leftarrow \texttt{bought}(\texttt{A},\texttt{B}),\texttt{cshares}(\texttt{B},\texttt{C},\_,\texttt{P}).$$

$$r3.\ \texttt{bought}(\texttt{A},\texttt{B}) \leftarrow \texttt{cshares}(\texttt{A},\texttt{B},\_,\texttt{P}),\texttt{P} > 50,\texttt{A} \neq \texttt{B}.$$

Traditional Database Management System recursive queries are efficiently implemented using $DeAL$. In the Company Control program proposed in [MPR90], companies can purchase ownership shares of other companies. In addition to the shares that company A owns directly, company A also controls the shares controlled by company B when A has a controlling majority ($> 50\%$) of B's shares.

Program 17. *What is the maximum cost of a part?*

$$r1.\ \texttt{cost}(\texttt{Part},\texttt{msum}\langle(\texttt{Part},\texttt{Cost})\rangle) \leftarrow \texttt{basic}(\texttt{Part},\_,\texttt{Cost}).$$

$$r2.\ \texttt{cost}(\texttt{Part},\texttt{msum}\langle(\texttt{Sub},\texttt{Cost})\rangle) \leftarrow \texttt{assb}(\texttt{Part},\texttt{Sub},\texttt{Num}),\texttt{cost}(\texttt{Sub},\texttt{Scost}),\ \texttt{Cost} = \texttt{Scost} * \texttt{Num}.$$

$$r3.\ \texttt{totalCost}(\texttt{Part},\texttt{max}\langle\texttt{Cost}\rangle) \leftarrow \texttt{cost}(\texttt{Part},\texttt{Cost}).$$

Program 17 is the Bill of Materials (BOM) program for finding the days required to deliver a part and the program for computing the max cost of a part from the cost of its subparts, respectively. The `assb` predicate denotes each part's required subparts and number required and `basic` denotes the number of days for a part to be received and the part's cost.

Program 18. *Viterbi Algorithm*

$$r1.\ \texttt{calcV}(0,\texttt{X},\texttt{mmax}\langle\texttt{L}\rangle) \leftarrow \texttt{s}(0,\texttt{EX}),\ \texttt{p}(\texttt{X},\texttt{EX},\texttt{L1}),\ \texttt{pi}(\texttt{X},\texttt{L2}),\texttt{L} = \texttt{L1} * \texttt{L2}.$$

$$r2.\ \texttt{calcV}(\texttt{T},\texttt{Y},\texttt{mmax}\langle\texttt{L}\rangle) \leftarrow \texttt{s}(\texttt{T},\texttt{EY}),\ \texttt{p}(\texttt{Y},\texttt{EY},\texttt{L1}),\ \texttt{T1} = \texttt{T} - 1,\texttt{t}(\texttt{X},\texttt{Y},\texttt{L2}),\ \texttt{calcV}(\texttt{T1},\texttt{X},\texttt{L3}),\texttt{L} = \texttt{L1} * \texttt{L2} * \texttt{L3}.$$

$$r3.\ \texttt{viterbi}(\texttt{T},\texttt{Y},\texttt{max}\langle\texttt{L}\rangle) \leftarrow \texttt{calcV}(\texttt{T},\texttt{Y},\texttt{L}).$$

Program 18 is the Viterbi algorithm for hidden Markov models. Four base predicates are used — t denotes the transition probability L2 from state X to Y; s denotes the observed sequence of length L+1; pi denotes the likelihood L2 that X is the initial state; p denotes the likelihood L1 that state X (Y) emitted EX (EY). *r1* finds the most likely initial observation for each X. *r2* finds the most likely transition for observation T for each Y. *r3* finds the max likelihood for T, Y.

Program 19. *Max Probability Path*

$$r1.\texttt{reach}(\texttt{X},\texttt{Y},\texttt{mmax}\langle\texttt{P}\rangle) \leftarrow \texttt{net}(\texttt{X},\texttt{Y},\texttt{P}).$$

$$r2.\texttt{reach}(\texttt{X},\texttt{Y},\texttt{mmax}\langle\texttt{P}\rangle) \leftarrow \texttt{reach}(\texttt{X},\texttt{Z},\texttt{P1}),\ \texttt{reach}(\texttt{Z},\texttt{Y},\texttt{P2}),\texttt{P} = \texttt{P1} * \texttt{P2}.$$

$$r3.\texttt{maxP}(\texttt{X},\texttt{Y},\texttt{max}\langle\texttt{P}\rangle) \quad \leftarrow \texttt{reach}(\texttt{X},\texttt{Y},\texttt{P}).$$

Program 19 is the non-linear program for computing the max probability path between two nodes in a network. The net predicate denotes the probability P of reaching Y from X.

Many Data Mining and Machine Learning applications use graphs as the underlying model, such as a Markov chain; and many graph analytics queries use a probability model. A Markov chain is represented by the transition matrix W of $s \times s$ components where w_{ij} is the probability to go from state i to state j in one step. A Markov chain is called *irreducible* if for each pair of states i, j, the probabilities to go from i to j and from j to i in one or more steps is greater than zero. Computing stabilized probabilities of a Markov chain has many real-world applications, such as estimating the distribution of population in a region, and determining the PageRank of web nodes. Let P be a vector of stabilized probabilities of cardinality s, the equilibrium condition in terms of matrices is: $P = W \cdot P$. Although computing this fixpoint is far from trivial, irreducible chains can be modeled quite naturally in $DeAL$. If $\texttt{p\_state}(\texttt{X},\texttt{K})$ denotes that K is the rank of node $\texttt{X}, 1 \leq \texttt{X} \leq \texttt{s}$, and $\texttt{w\_matrix}(\texttt{Y},\texttt{X},\texttt{W})$ denotes that there is an arc from Y to X with weight W. Then, we compute the fixpoint using Program 20.

Program 20. *Markov Chains*

$r1$. $\texttt{p\_state}(\texttt{X}, \texttt{mmax}\langle\texttt{P}\rangle) \leftarrow \texttt{p\_state\_init}(\texttt{X}, \texttt{P})$.

$r2$. $\texttt{p\_state}(\texttt{X}, \texttt{msum}\langle(\texttt{Y}, \texttt{K})\rangle) \leftarrow \texttt{p\_state}(\texttt{Y}, \texttt{C}), \texttt{w\_matrix}(\texttt{Y}, \texttt{X}, \texttt{W}), \texttt{K} = \texttt{C} * \texttt{W}$.

$r3$. $\texttt{rank}(\texttt{X}, \texttt{max}\langle\texttt{K}\rangle) \leftarrow \texttt{p\_state}(\texttt{X}, \texttt{K})$.

$r4$. $\texttt{sum\_rank}(\texttt{sum}\langle\texttt{A}\rangle) \leftarrow \texttt{rank}(\_, \texttt{A})$.

$r5$. $\texttt{p\_norm}(\texttt{X}, \texttt{Pr}) \leftarrow \texttt{sum\_rank}(\texttt{SR}), \texttt{rank}(\texttt{X}, \texttt{R}), \texttt{Pr} = \texttt{R}/\texttt{SR}$.

Note that each fixpoint of such a program is an equilibrium $P = W \cdot P$ of the Markov Chain represented by matrix W. In order to find a non trivial fixpoint ($\neq 0$) for program P, we start with facts such as $\texttt{p\_state\_init}(1, 0.1)$., $\texttt{p\_state\_init}(2, 0.1)$, ..., $\texttt{p\_state\_init}(\texttt{s}, 0.1)$, which provide the baseline set facts of the least fixpoint T_{Pbl}. For each irreducible Markov chain there exists a non trivial fixpoint, therefore T_P has one that is not null at every node, and there exists a finite fixpoint for T_{Pbl}. We compute this fixpoint using *r1* and *r2*. *r3* produces only the maximum probability for each node X. Then rules $r4$ and $r5$ normalize the probabilities to produce the final result.

4.14 Syntax Comparison With Other Languages

Here we present examples showing the syntactic differences between using $DeAL$'s monotonic aggregates and other implemented approaches. The example programs shown compute the shortest paths on a directed graph. Recall, the $DeAL$ program using monotonic aggregates for Shortest Paths is Program 3. In the interest of space, we omit structural declarations and rules to establish the 0 cost path between a vertex and itself. With the exception of Program 21, the programs will terminate on cyclic graphs with positive cost edge weights.

Program 21. *Stratified Shortest Paths*

$r1$. $\texttt{spaths}(\texttt{X}, \texttt{Y}, \texttt{D}) \leftarrow \texttt{edge}(\texttt{X}, \texttt{Y}, \texttt{D})$.

$r2$. $\texttt{spaths}(\texttt{X}, \texttt{Z}, \texttt{D}) \leftarrow \texttt{spaths}(\texttt{X}, \texttt{Y}, \texttt{D1}), \texttt{edge}(\texttt{Y}, \texttt{Z}, \texttt{D2}), \texttt{D} = \texttt{D1} + \texttt{D2}$.

$r3$. $\texttt{shortestpaths}(\texttt{X}, \texttt{Z}, \texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{spaths}(\texttt{X}, \texttt{Z}, \texttt{D})$.

Program 21 is a $DeAL$ program that first computes the transitive closure of the graph with the cost of each path (*r1*, *r2*) and then uses a stratified `min` aggregate to find the shortest path between two vertices (*r3*).

Program 22. *XY-Stratified Shortest Paths*

$r1.\ \texttt{delta\_sp}(0, \texttt{X}, \texttt{Y}, \texttt{C}) \leftarrow \texttt{edge}(\texttt{X}, \texttt{Y}, \texttt{C}).$

$r2.\ \texttt{delta\_sp}(\texttt{J+1}, \texttt{X}, \texttt{Z}, \texttt{min}\langle\texttt{C}\rangle) \leftarrow \texttt{delta\_sp}(\texttt{J}, \texttt{X}, \texttt{Y}, \texttt{C1}), \texttt{edge}(\texttt{Y}, \texttt{Z}, \texttt{C2}), \texttt{C=C1+C2},$
$\texttt{if}(\texttt{all\_sp}(\texttt{J}, \texttt{X}, \texttt{Z}, \texttt{C3})\ \texttt{then}\ \texttt{C3} > \texttt{C}).$

$r3.\ \texttt{all\_sp}(\texttt{J+1}, \texttt{X}, \texttt{Z}, \texttt{C}) \leftarrow \texttt{all\_sp}(\texttt{J}, \texttt{X}, \texttt{Z}, \texttt{C}), \neg\texttt{delta\_sp}(\texttt{J+1}, \texttt{X}, \texttt{Z}, \_).$

$r4.\ \texttt{all\_sp}(\texttt{J}, \texttt{X}, \texttt{Z}, \texttt{C}) \leftarrow \texttt{delta\_sp}(\texttt{J}, \texttt{X}, \texttt{Z}, \texttt{C}).$

$r5.\ \texttt{lastsp}(\texttt{I}) \leftarrow \texttt{delta\_sp}(\texttt{I}, \_, \_, \_), \neg\texttt{delta\_sp}(\texttt{I+1}, \_, \_, \_).$

$r6.\ \texttt{shortestpaths}(\texttt{X}, \texttt{Y}, \texttt{C}) \leftarrow \texttt{lastsp}(\texttt{I}), \texttt{all\_sp}(\texttt{I}, \texttt{X}, \texttt{Y}, \texttt{C}).$

Program 22 is the XY-Stratified $DeAL$ program. XY-Stratified programs [AOT03, ZAO93] are executed using an iterative state machinery that processes facts across two states (J, J+1) and can therefore support a wide range of algorithms, but can suffer from inefficient performance. The intuition for Program 22 is after each iteration, a stratified min aggregate is used to select the new shortest paths (delta_sp) and the non-shortest paths from earlier iterations are discarded. all_sp represents all shortest paths found and when no new shortest paths are found, the program terminates.

In Program 22, *r1* copies all edge facts into the initial state of delta_sp. *r2* joins the paths in delta_sp with edge to find new paths shorter than paths already in all_sp[18]. *r2* also applies a stratified min aggregate to find the shortest paths between two vertices discovered at state J to bring to the next state J+1. *r3* says that paths in all_sp at state J+1 consist of paths from all_sp at state J except for new shortest paths just found (delta_sp) because these are shorter, and the longer paths already in all_sp must be deleted. The expressive power of XY-Stratified rules comes at the expense of a larger program and having to specify procedural-like rules such as *r4*, which copies facts between relations, *r5* which finds the final iteration at which a fixpoint is reached, and *r6* which retrieves the results from the final state.

Program 23. *Shortest Paths in LogiQL*

$r1.\ \texttt{path}(\texttt{x}, \texttt{y}, \texttt{d}) \leftarrow \texttt{edge}(\texttt{x}, \texttt{y}, \texttt{d});$
$\texttt{edge}(\texttt{x}, \texttt{z}, \texttt{d1}), \texttt{spath}[\texttt{z}, \texttt{y}] = \texttt{d2}, \texttt{d} = \texttt{d1} + \texttt{d2}.$

$r2.\ \texttt{spath}[\texttt{x}, \texttt{y}] = \texttt{dist} \leftarrow \texttt{agg}\langle\langle\texttt{dist} = \texttt{min}(\texttt{d})\rangle\rangle\ \texttt{path}(\texttt{x}, \texttt{y}, \texttt{d}).$

[18]The conditional if (then) predicate is the logical equivalent of two rules, one with and one without negation.

Program 23 is the LogicBlox's LogiQL program to compute the shortest path in a directed graph. LogicBlox has recently included support for aggregation in recursion [19]. This program uses what they term a *staged partial fixpoint semantics* and syntactically exhibits a mutual recursion between *r1*, which generates the paths, and *r2* which applies the `min` aggregate to find the shortest path between two vertices.

Program 24. *Shortest Paths in Bloom^L*

```
r1.path <= edge{|e|[e.from,e.to,MinLattice.new(e.c)]}
r2.path <= (path * edge).pairs(:to => :from) do |p,e|
    [p.from, e.to, p.c + e.c]
  end
r3.shortestpaths <= path {|p| [p.from, p.to, p.c]}
```

Program 24 is the Bloom^L [CMA12] program for shortest paths which utilizes the `MinLattice` lattice type to find the minimum cost path between two vertices. *r1* identifies `edge` facts as `path` facts using the MinLattice to track the minimum cost for each pair of vertices. *r2* the generates new shortest paths by concatenating `edge` facts with existing `path` facts. Although designed for distributed programming, the semantics of the lattice types used in Bloom^L share many similarities with the semantics proposed in [RS92]. We refer the reader to [CMA12] for more details on lattice types in Bloom^L.

Discussion. From the examples in Section 4.14, the reader should observe that Program 3, the $DeAL$ program using monotonic aggregates, pushes the aggregation inside the recursion. This is particularly noticeable when comparing Program 3 with Example 21. The monotonic aggregate is more concise than the XY-Stratified program (Program 22). From a syntactic point of view, the LogiQL program is concise but iterates over building a set of paths and then pruning suboptimal paths from the set. This is more similar to the XY-stratified program than to the monotonic aggregate approach as the monotonic aggregates only build the set of paths and refine the costs. The

[19] Found in LogicBlox 4.0 migration guide https://download.logicblox.com/content/docs4/release-notes/4-0-Migration/pdf/LB-MigrationGuide-40.pdf. We renamed predicate `e` to `edge` to match the other programs.

Bloom^L program is concise and the syntax requires only specifying the type of the cost argument in the base case of the program.

4.15 Additional Related Work

We have previously discussed the contributions of many works on supporting aggregation in recursion including [SGL13, RS92, FPL08]. For extrema aggregates, [GGZ91] proposes rewriting programs by pushing the aggregate into the recursion and *Greedy Fixpoint* evaluation to select the next min/max value to execute. Another approach proposes using *aggregate selections* to identify *irrelevant* facts that are to be discarded by an extended version of SN [SR91]. Nondeterministic constructs and stable models have been proposed to define monotone aggregates [ZW99]. The DRed [GMS93] algorithm incrementally maintains recursive views that can include negation and aggregation. The Bloom^L distributed programming language [CMA12] uses logical monotonicity via built-in and user-defined lattice types to support eventual consistency.

4.16 Monotonic Aggregates Summary

In this chapter, we have shown how monotonicity can be extended to the aggregate involved in recursive computations while preserving the syntax, semantics, and optimization techniques of traditional Datalog. The significance of this result follows from the fact that this problem had remained long unsolved, and that many new applications can be expressed with the proposed extensions that make them amenable to parallel execution on multiprocessor and distributed systems.

CHAPTER 5

BigDatalog - DeAL on Apache Spark

In this chapter, we extend $DeALS$ for distributed evaluation to support advanced analytics over massive datasets. To support efficient distributed DATALOG evaluation, we must utilize a distributed engine designed for iteration. Towards this end, we select Apache Spark [apa15b] as our execution engine. Spark 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. By utilizing Spark we are attempting to maximize the potential usage of $DeAL$ and the impact of the lessons we learned.

One could expect that as a system designed for iterative applications, Apache Spark would also be well suited for recursive applications such as shortest paths computations, and link and graph structure analysis. However this ignores three decades of recursive query evaluation and optimization techniques. Spark's support of recursion through iteration is inefficient: in an iterative Spark application, a new job is submitted for every iteration and thus the system has only limited visibility over an application's entire execution. From a programming perspective, the development of efficient recursive applications in Spark requires the programmer to have a deep understanding of (1) the algorithm being implemented, (2) the Spark API, and (3) Spark internals. Nevertheless, Spark is a promising system for recursive applications because it provides many features essential for recursive evaluation, including dataset caching and low task startup costs. Along those lines, as we implement BigDatalog, we examine how Spark can be made to efficiently support recursive applications.

BigDatalog is a full DATALOG language implementation on Spark. BigDatalog provides logical abstraction and enables the concise and declarative expression of relational and recursive queries

while maintaining an efficient execution. BigDatalog enables a host of declarative language optimization techniques and it exploits semantic extensions for programs with *aggregation in recursion* [MSZ13a, MSZ13b] and programs that can be *evaluated without communication* [SL91]. Furthermore, since BigDatalog supports relational algebra, aggregation and recursion, the Spark programmer can now implement complex analytics pipelines of relational, graph and machine learning tasks in a single language instead of having to stitch together programs written in different APIs of Spark subsystems such as SparkSQL [AXL15] and GraphX [GXD14].

Motivating Example. As an example of the performance improvement that BigDatalog achieves for the evaluation of recursive queries consider Figure 5.1. This figure shows the execution time required to compute a 100 million vertex pair transitive closure of a graph using a highly optimized handwritten Spark program versus the BigDatalog version (cluster specs. in Section 5.5). This example shows how BigDatalog is both considerably better than its host framework and also performs quite well when compared with large-scale DATALOG systems, namely Myria [WBH15] and SociaLite [SPS13] in our test environment. This order of magnitude speed-up is achieved by employing the efficient evaluation techniques and optimizations of DATALOG in Spark.

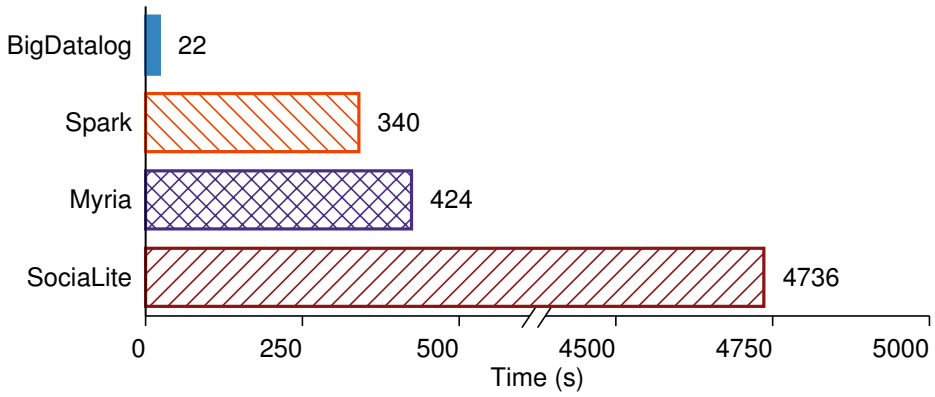


Figure 5.1: Example Recursive Query Performance.

Contributions and Outline. In this chapter we make the following contributions:

- We design and implement the BigDatalog compiler. We show how BigDatalog programs are compiled into recursive physical plans for Spark.

- We present a parallel evaluation technique for DATALOG evaluation on Spark. We introduce recursion operators and data structures to efficiently implement the technique on Spark.
- We propose physical planning and scheduler optimizations for recursive queries on Spark.
- We present distributed monotonic aggregates, an accompanying evaluation technique, and data structures to support DATALOG programs with aggregates on Spark.

Outline. In Section 5.1, we review Spark. Section 5.2 introduces BigDatalog and the distributed evaluation technique used to evaluate BigDatalog programs, and shows how BigDatalog programs are compiled into physical plans for execution on Spark. Section 5.3 presents evaluation, physical plan and job scheduling optimizations. Section 5.4, describes our aggregate design and implementation. Section 5.5 presents experimental results including comparisons of BigDatalog with other large-scale DATALOG systems. Section 5.6 reviews related works. Section 5.7 presents a summary for BigDatalog and plans for future work.

5.1 Preliminaries

In this section, we review Spark and discuss the challenges of using Spark as a DATALOG runtime.

5.1.1 Apache Spark

Spark provides a language-integrated Scala API enabling the expression of programs as *dataflows of transformations* (e.g., `map`, `filter`) on *Resilient Distributed Datasets* (RDD) [ZCD12]. An RDD is a distributed shared memory abstraction representing a partitioned dataset; RDDs are immutable, and transformations are coarse-grained and thus apply to all items in the RDD to produce a new RDD. Spark executes transformations lazily: a *job* is submitted for execution only when *actions* such as `count` or `reduce` are called by the user's program. Once a job is submitted, the scheduler groups transformations that can be pipelined (e.g., `map` over a `join`) into a single *stage*. The stages composing a dataflow are executed *synchronously* in a topological order: a stage will not be scheduled until all stages it is dependent upon have finished successfully. Between

stages, Spark *shuffles* the dataset to *repartition* it among the nodes of the cluster. When a stage can be run, the scheduler creates a set of *tasks* (i.e., execution units) consisting of one task for each input *partition*, and launches the tasks on *worker nodes*.

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). Spark has libraries for structured data processing (SparkSQL), streaming [ZDL13], machine learning (MLlib), and graph processing (GraphX).

SparkSQL. Spark's structured data and relational processing module, supports a subset of SQL. SparkSQL provides logical and physical relational operators. SparkSQL physical operators use a pipelined iterator model and are implemented as functions applied over the iterator from an upstream operator. The Catalyst framework [AXL15] supports the compilation and optimization of SparkSQL programs into physical plans. In this work, we use and extend SparkSQL operators. We also propose BigDatalog operators that are implemented using the Catalyst framework so BigDatalog can use Catalyst planning features on recursive plans.

Iterative Spark Programs. Spark iterative applications are implemented by having a *driver program* iterate over a sequence of transformations terminated by an action. Each iteration is a new job that operates on cached RDD(s) produced by the previous iteration. Iteration terminates after a user-defined number of iterations or based on a user-defined predicate that determines when convergence has been reached. Examples of algorithms supported by this approach include PageRank, logistic regression, and the Semi-Naïve TC shown in Figure 5.2.

Figure 5.2 is explained as follows. After some initial setup including distributing the graph among nodes of the cluster (line 1) and preparing the edges of the graph for joins (line 2), the program enters a `do-while` loop. It will iterate by executing a new job for each `count` action, until an iteration produces no new results (line 9). In each iteration, the program will join facts from the previous iteration (`deltaTC`) with `arcs` (line 5), project the pair of vertices (line 6), and eliminate duplicates (line 7). The set of all previously produced pairs is then combined with the newly produced pairs (line 8). Reused RDDs are cached (lines 2, 7, 8).

```
1  var tc = sc.parallelize(graph, numPartitions)
2  val arcs = tc.map(x => (x._2, x._1)).cache()
3  var deltaTC = tc
4  do {
5    deltaTC = deltaTC.join(arcs)
6      .map(x => (x._2._2, x._2._1))
7     .subtract(tc).distinct().cache()
8    tc = tc.union(deltaTC).cache()
9  } while (deltaTC.count() > 0)
10 tc
```

Figure 5.2: Semi-Naïve TC Spark Program.

Note the simplicity of the DATALOG program in Program 1 compared to the Spark program in Figure 5.2. Spark requires the programmer (1) be familiar with semi-naïve evaluation, (2) directly express a dataflow's physical plan composed of properly ordered operations and (3) handle memory management (RDD caching) to obtain better performance. Instead, BigDatalog enables high-level specification amenable to optimizations and rescues the programmer from extensive coding, debugging and maintenance effort.

5.1.2 Challenges for Datalog on Spark

Now that we have provided background on Spark, we can discuss the three main challenges we face with implementing DATALOG in Spark.

1. Acyclic Plans: Supporting compilation, optimization and evaluation of DATALOG programs on Spark requires features not currently supported. A recursive, rule-based syntax requires a different compiler front-end than SparkSQL language queries. SparkSQL lacks recursion operators, operators are designed for acyclic use, and the Catalyst optimizer is targeted for non-recursive plans.

2. Scheduling: Spark uses a synchronous stage-based scheduler that issues tasks for a stage only

after all tasks of the previous stages have completed. For (monotonic) DATALOG programs, like the ones studied in this chapter, this can be seen as unnecessary coordination because monotonic DATALOG programs are eventually consistent [ANB11].

3. RDD Immutability & Memory Utilization: Each iteration of recursion will produce a new RDD representing the facts obtained thus far during evaluation of the recursive predicate. This RDD will contain both new facts and all the facts produced in earlier iterations, which are already contained in earlier RDDs. If poorly managed, recursive applications on Spark can experience memory utilization problems.

5.2 BigDatalog

BigDatalog is a distributed DATALOG language implementation for analytics. With BigDatalog, programs are expressed as DATALOG rules, then compiled, optimized and executed on Spark. BigDatalog will manage the persistence of datasets and make partitioning decisions. BigDatalog uses set-semantics and supports recursion, non-monotonic aggregation (`count`, `sum`, `min`, `max`, `avg`) and aggregation in recursion with monotonic aggregates (cf. Section 5.4).

5.2.1 Benchmark Programs

In this chapter, we focus on monotonic (positive) programs which include classical recursive queries from the literature as well as aggregate queries, some of which are long studied (e.g., shortest paths) and others studied more recently (connected components) [SPS13, WBH15].

Classical Recursive Queries

- **Transitive Closure** (TC) (Program 1)
- **Same Generation** (SG) identifies pairs of vertices where both are the same number of hops from a common ancestor.
- **Reachability** (REACH) produces all nodes connected by some path to a given source node.

Aggregation in Recursion Queries

- **Single-Source Shortest Paths** (SSSP) computes the length of the shortest path from a source vertex to each vertex it is connected to.
- **Connected Components** (CC) identifies connected components in the graph.

The programs selected can be separated into two categories. The first category includes programs such as TC and SG which produce large result sets, even from small graphs. For TC and SG we will use graphs that are relatively small in terms of number of vertices or edges. The second category includes programs such as REACH, CC, and SSSP that produce result sets smaller than the input graphs, and so for this category we experiment with large graphs. Next, we show how Program 1 is executed on Spark using the BigDatalog API.

5.2.2 BigDatalog API By Example

The program snippet shown in Figure 5.3 computes the size of the transitive closure of the graph using the BigDatalog API for Spark. In a driver program, 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 database schema definition for base relations and program rules (lines 2-4). Lines 3-4 implement TC from Program 1. The database definition and rules are given to the BigDatalog compiler which loads the database schema into a relation catalog (line 5). Next, the data source path (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).

5.2.3 Parallel Semi-Naïve Evaluation on Spark

BigDatalog programs are evaluated using a parallel version of SN 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, where an iteration will not begin until all tasks from the previous iteration have

```
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 5.3: BigDatalog Program for Spark.

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 framework. Plans are composed of SparkSQL and BigDatalog operators. A plan produces an RDD representing the plan's transformations. The RDD for the exit rules plan is first evaluated once. Then, each iteration until a fixpoint is reached, the recursive rules plan is used to produce a new RDD, which is evaluated to produce the facts for the iteration. This design allow BigDatalog to apply the same sequence of transformations over different RDDs each iteration without the cost of re-compilation, logical planning or physical planning. Lastly, note that as with SN, PSN will also evaluate symbolically rewritten rules (e.g., $\texttt{tc}(\texttt{X},\texttt{Y}) \leftarrow \delta\texttt{tc}(\texttt{X},\texttt{Z}),\ \texttt{arc}(\texttt{Z},\texttt{Y})$.).

Algorithm 5 is the psuedocode for the PSN using RDDs and is explained as follows. The **exitRulesPlan** (line 1) and **recursiveRulesPlan** (line 5) are plans for the exit rules and recursive rules, respectively. We use `toRDD()` (lines 1,5) 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 3,8) 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

Algorithm 5 PSN Framework with RDDs - Psuedocode

```
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
```

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 5 to simplify its presentation. Lastly, in cases of mutual recursion, when two or more rules belonging to different predicates reference each other (e.g., A $\leftarrow$ B, B $\leftarrow$ A), one predicate will "drive" the recursion with PSN and the other recursive predicate(s) will be an operator in the driver's recursive rules plan.

5.2.4 Compilation and Planning

The BigDatalog compiler supports the parallel, distributed bottom-up evaluation of DATALOG programs. Figure 5.4 depicts the compilation workflow for BigDatalog programs. The input for the compiler is a database schema definition, a set of rules and a query. From this, the compiler creates a logical plan for the program, which is optimized using database techniques such as projection

pruning. The logical plan for a non-recursive BigDatalog query is mapped into a SparkSQL plan and executed accordingly. However, logical plans for recursive queries are converted to BigDatalog physical plans.

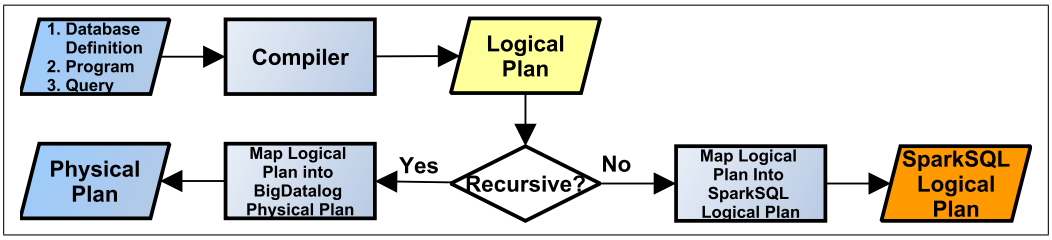


Figure 5.4: BigDatalog Compilation Workflow.

5.2.4.1 Logical Plans

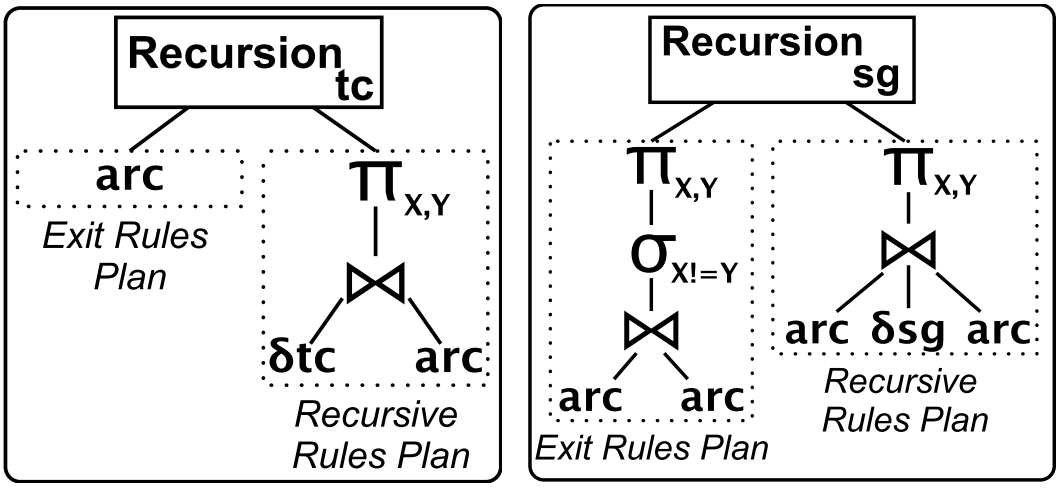
Here, we use Program 1 (TC) to describe how the compiler produces a logical plan. Given the query `tc(X,Y)`, the program is first compiled into a Predicate Connection Graph (PCG) (Section 3) to identify the exit rules (*r1*) and recursive rules (*r2*) of the `tc` recursive predicate. From the PCG, the logical query plan is produced by mapping it into a tree of relational and recursion (e.g., SN) operators. A recursion operator has two child logical (sub)plans: one plan for the predicate's exit rules and the other for the predicate's recursive rules. Figure 5.5(a) is the logical plan produced by the BigDatalog compiler for Program 1. The left side of Figure 5.5(a) is the exit rules plan with only the `arc` relation, representing *r1*. The right side of Figure 5.5(a) is the recursive rules plan made up of relational operators to produce one iteration of *r2*.

Program 25. *Same Generation*

$$r1.\ \mathtt{sg(X,Y)} \leftarrow \mathtt{arc(P,X)},\ \mathtt{arc(P,Y)},\ \mathtt{X} \mathtt{!} = \mathtt{Y}.$$
$$r2.\ \mathtt{sg(X,Y)} \leftarrow \mathtt{arc(A,X)},\ \mathtt{sg(A,B)},\ \mathtt{arc(B,Y)}.$$

Consider Program 25, the Same Generation (SG) program. 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. For PSN, *r2* is (symbolically) rewritten as $\mathtt{sg(X,Y)} \leftarrow \mathtt{arc(A,X)},\ \delta\mathtt{sg(A,B)},\ \mathtt{arc(B,Y)}$. The left side of Figure 5.5(b) is the exit rules plan

with a self-join of `arc` to find siblings. The right side of Figure 5.5(b) is the recursive rules plan which includes a three-way join of δ`sg` and `arc`.



(a) TC (b) SG

Figure 5.5: BigDatalog Logical Plans.

5.2.4.2 Physical Plans

BigDatalog translates logical plans into physical plans comprised of SparkSQL and BigDatalog physical operators. Like SparkSQL operators, BigDatalog operators use the SparkSQL `Row` type. Most logical-to-physical operator mapping is straightforward, however *recursion*, *join* and *shuffle* operators require discussion.

Recursion Operators. The *Recursion Operator* (RO) is a special driver operator that runs on the master and executes PSN, i.e., the psuedocode from Algorithm 5. An RO has two child physical (sub)plans, the *Exit Rules Plan* (ERP) and the *Recursive Rules Plan* (RRP). A *Recursive Relation* operator represents a recursive predicate in the RRP and produces the recursive relation when evaluated (i.e., the plan version of a recursive predicate body literal).

Join Operators. BigDatalog uses binary hash join operators where one input is loaded into a lookup hashtable and tuples of the other input are streamed; the hash of the tuple's join key attributes is used to probe the lookup table for matches. Since rules, PCG and logical plans all

support multi-way joins, but physical join operators are binary operators, we convert a multi-way logical join operator into a hierarchy of binary physical join operators, in a left-to-right fashion.

In a linear recursion, where only one join input is a recursive relation, the non-recursive input is loaded into lookup tables and the recursive relation is streamed. For instance, from the logical plan for TC in Figure 5.5(a), the RRP will have a join where δtc is streamed and arc is loaded into lookup tables. To help explain our approach for non-linear recursions, we use the following non-linear program. In this program, *r2* creates new tc facts of the form $(\texttt{X}, \texttt{Y})$ by joining tc facts of the form $(\texttt{X}, \texttt{Z})$ with tc facts of the form $(\texttt{Z}, \texttt{Y})$.

Program 26. *Non-Linear Transitive Closure*

$$r1.\texttt{tc}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{arc}(\texttt{X}, \texttt{Y}).$$
$$r2.\texttt{tc}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{tc}(\texttt{X}, \texttt{Z}),\ \texttt{tc}(\texttt{Z}, \texttt{Y}).$$

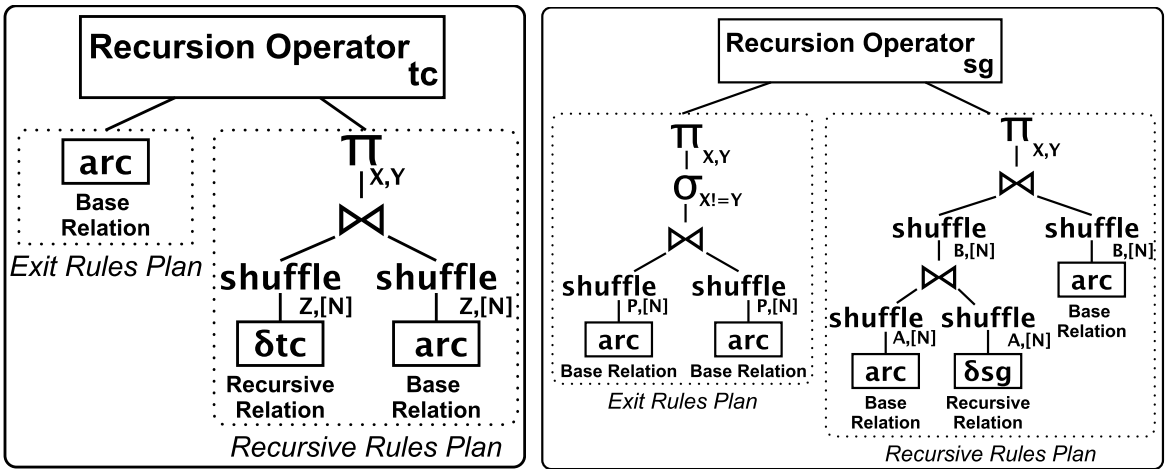
In Program 26, *r2* will be (symbolically) rewritten for SN, as:

$$\texttt{tc}(\texttt{X}, \texttt{Y}) \leftarrow \delta\texttt{tc}(\texttt{X}, \texttt{Z}),\ \texttt{tc}(\texttt{Z}, \texttt{Y}).$$

Since both inputs to the join are recursive relations, δtc will be loaded into lookup tables and tc will be streamed. We choose this approach because loading the smaller of the two into lookup tables is less expensive and after a few iterations, tc is likely to be much larger than δtc. Program 26 performs the logarithm of the number of iterations of the linear TC Program 1.

Shuffle Operators. After mapping the logical operators into physical operators, the last step to produce a physical plan for execution is to add *shuffle operators* for distributed evaluation. Shuffle operators are used to repartition the dataset when there is a mismatch between an operator's required input partitioning and a child operator's output partitioning. For example, a shuffle operator is needed to repartition an input to a join if the input is not partitioned on the join keys. We use a Catalyst feature to analyze the physical plan and add shuffle operators where needed. We use hash partitioning and a static number of partitions throughout evaluation. Future work is to investigate dynamically adjusting the number of partitions during evaluation.

Example Plans. The physical plans produced for Program 1 (TC) and Program 25 (SG) are displayed in Figure 5.6(a) and 5.6(b), respectively. Using Figure 5.6(a) as our point of reference,



(a) TC (b) SG

Figure 5.6: BigDatalog Physical Plans.

we explain how our operators from above are used in plans. The root of the plan is the RO for the `tc` recursive predicate. In the RRP, δ`tc` is a recursive relation and when evaluated will produce `tc`'s facts from the previous iteration. Both inputs to the hash join are shuffled. The subscript `Z`, [`N`] indicates the partitioning key is the `Z` argument (from the program rule), and there will be `N` partitions. Here `Z` is the join argument so that tuples of `arc` and δ`tc` having the same key will be co-located on the same worker.

Figure 5.7 depicts the RDD lineage graph produced when evaluating the physical plan shown in Figure 5.6(a) up through two iterations of PSN. Mapping the evaluation of Figure 5.6(a) with PSN into RDDs is straightforward, however some important points are the following. The reader should notice the box with ERP (RRP) in the corner encloses the ERP (RRP) RDDs and transformations. RDDs (and transformations) not enclosed in boxes are produced by the PSN framework. We use a dashed line to indicate $\texttt{tc}_0$ is the same RDD as $\delta\texttt{tc}_0$. Evaluation begins by reading in an input file and partitioning it via `coalesce` into a user-defined number of partitions (e.g., number of cpu cores in the cluster) to create the `arc` RDD. Lastly, note that in addition to the shuffle operations prior to the join of `arc` and δ`tc`, the `subtract` and `distinct` transformations applied by PSN also shuffle.

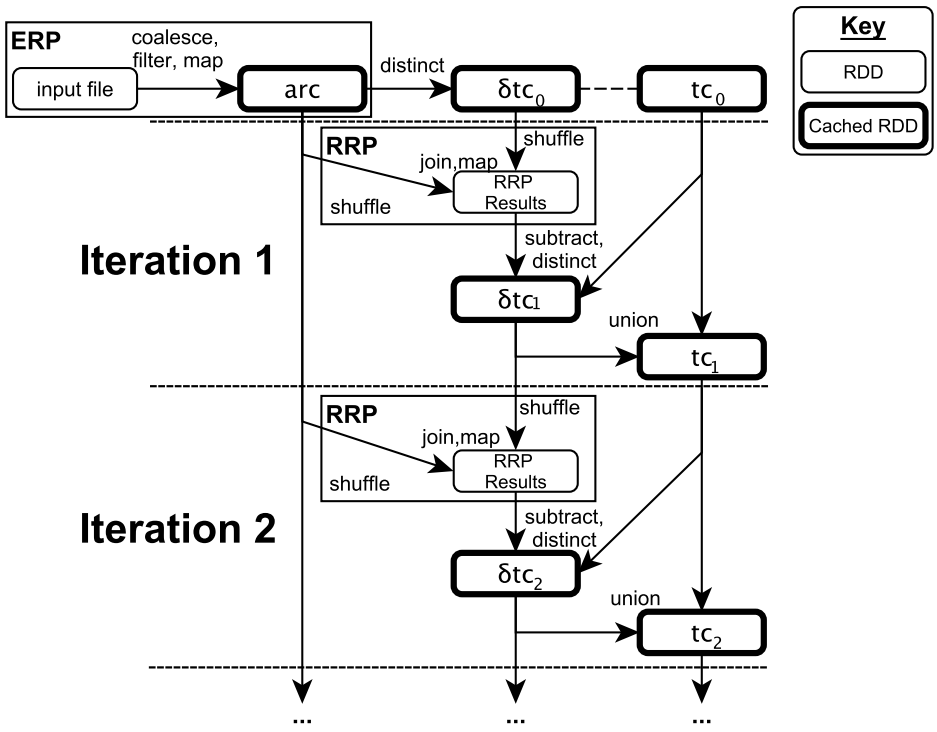


Figure 5.7: RDD Lineage Graph for TC Physical Plan.

5.3 Optimizations

This section presents optimizations to improve the performance of BigDatalog programs. Details on the datasets used in experiments in this section can be found in Section 5.5.

5.3.1 Optimizing PSN

As shown with Algorithm 5, PSN can be implemented with RDDs and transformations such as `subtract`, `distinct` and `union`. However, using standard RDD transformations is inefficient because each iteration the results of the recursive rules are set-differenced with the entire recursive relation (line 6 in Algorithm 5), which is growing each iteration. Since an RDD stores facts in an array, expensive data structures must be created each iteration. We propose instead to use the *SetRDD*, a specialized RDD for storing distinct `Rows` and tailored for set operations needed for

PSN. Each partition of SetRDD is a single set data structure.[1] By storing facts in a set, SetRDD efficiently supports set-difference and de-duplication operations. By caching SetRDD, the need to rebuild expensive data structures each iteration is eliminated. SetRDD methods include `union`[2], which unions two SetRDDs to produce a new SetRDD, and `diff`, which produces a new SetRDD containing rows from the input RDD not already in the SetRDD. Converting PSN to use SetRDD is straightforward.

```
class SetRDD(partitionsRDD: RDD[SetRDDPartition[Row]])
  extends RDD[Row] {
  def diff(other: RDD[Row]): SetRDD
  override def union(other: RDD[Row]): SetRDD
...}
abstract class SetRDDPartition[T] {
  def diff(iter: Iterator[T]): SetRDDPartition[T]
  def union(other: SetRDDPartition[T]): SetRDDPartition[T]
...}
object SetRDD {
  def apply(rdd: RDD[Row], schema: Schema): SetRDD = {
    new SetRDD(rdd.mapPartitions[SetRDDPartition[Row]] (
      iter => Iterator(SetRDDHashSetPartition(iter, schema)),
       true))
}
```

Figure 5.8: SetRDD Interface.

In SetRDD, facts are efficiently stored using type-specific data structures to store primitive types sufficient to capture the schema or, in the case of larger schema, `Row` objects themselves. For instance, the recursive relation of either TC or SG will store pairs of vertices with a schema of two 4-byte integers. We combine the two 4-byte integers into one 8-byte integer (`long`) and use a

[1] This design is inspired by GraphX's specialized RDD for vertices (`VertexRDDImpl`) which stores multiple data structures (e.g., bitset, array) in each RDD partition.

[2] This is a set union i.e., it removes duplicates.

`HashSet` from fastutil [fas15] optimized for `long`. Storing primitives saves memory and garbage collections as compared to storing more expensive wrapper types (e.g., `java.lang.Long`) and provides fast existential checks for de-duplication. SetRDD can be configured to also utilize specialized B+trees for `int`, `long`, and `byte` arrays (for larger schema). `Row` objects are produced from the primitive types by a schema-aware iterator.

Figure 5.8 displays the interfaces for the `SetRDD` and `SetRDDPartition` classes and Figure 5.9 displays the implementation of `SetRDDHashSetPartition`. Note the `HashSet`-typed argument in the constructor to `SetRDDHashSetPartition` - this is a type-specific HashSet structure and has methods `size`, `insert(Row)` and `contains(Row)`. The design of `SetRDDBPlusTreePartition` is similar and uses a B+Tree instead of HashSet.

Monotonicity and RDD Immutability. We can apply an optimization enabled by DATALOG set-containment semantics to efficiently produce a new SetRDD for the union transformation. Although an RDD is intended to be immutable, we make SetRDD mutable under the union operation. From `union`, a new SetRDD will be produced with partitions referencing the same HashSets as its creator. For example, in Figure 5.9 both `union` methods add facts to the partition's existing HashSet. In the event that a task performing the union fails and must be re-executed, it will not lead to incorrect results because union is monotonic and facts can be added only once to a relation. This design saves system memory because only one HashSet exists per partition across all iterations.

Table 5.1: PSN vs. PSN with SetRDD Performance

Time (s)	TC			SG		
	`Tree17`	`Grid150`	`G10K`	`Tree11`	`Grid150`	`G10K`
PSN	244	OOM	208	OOM	230	1129
PSN with SetRDD	41	134	20	59	61	130

Table 5.1 displays the results of evaluating TC and SG with both PSN and PSN with SetRDD. PSN with SetRDD outperforms PSN significantly in all cases.

```
class SetRDDHashSetPartition(set: HashSet, schema: Schema)
  extends SetRDDPartition[Row] {
  override def union(otherPartition: SetRDDPartition[Row]):
      SetRDDHashSetPartition = {
    otherPartition match {
      case otherPartition: SetRDDHashSetPartition => {
        new SetRDDHashSetPartition(
              set.union(otherPartition.set), schema)
      }
      case other => union(otherPartition.iterator)
    }
  }
  override def union(iter: Iterator[Row]):
      SetRDDHashSetPartition = {
    while (iter.hasNext)
      set.insert(iter.next())
    new SetRDDHashSetPartition(set, schema)
  }
...}
```

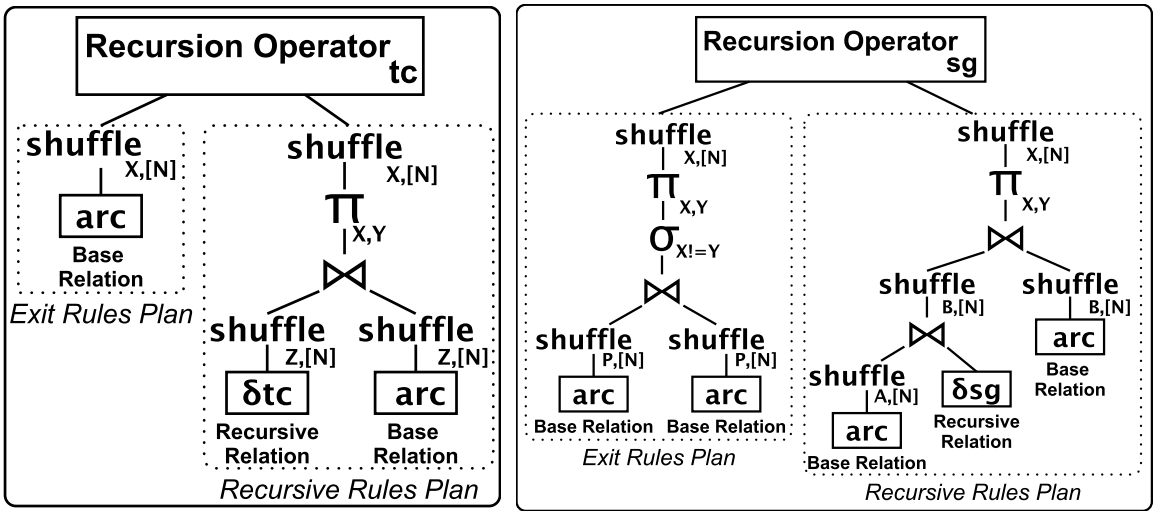
Figure 5.9: SetRDDHashSetPartition Implementation.

5.3.2 Partitioning

The initial version of PSN used RDD transformations (e.g., `distinct`, `subtract`) that performed the necessary shuffling operations. That approach was sufficient to produce a correct result, but could be inefficient to evaluate. Now, SetRDD's `diff` and `union` transformations are designed to require properly partitioned input (i.e., they will not shuffle). Therefore, none of the transformations used in PSN will repartition inputs so shuffle operators need to be placed into ERP and RRP to produce properly partitioned output for PSN transformations. This approach allows for a simplified and generalized PSN framework and brings the insertion of shuffle operators to

the workflow under the control of the BigDatalog compiler. With full control over shuffle operator placement, (i.e., communication decisions), BigDatalog can produce very efficient evaluations.

Earlier DATALOG research showed a good *partitioning strategy* (i.e., the arguments on which to partition) for a recursive predicate was important for efficient parallel evaluation [CW89, GST90, GST92, WO90]. In general, we seek a partitioning strategy that limits shuffling. The default partitioning strategy employed by BigDatalog is to partition the recursive predicate on the *first argument*. Now we can produce plans for PSN that will terminate with a shuffle operator if the output of the plan does not match the partitioning strategy of the predicate. Figure 5.10(a) is the plan for Program 1 for PSN with SetRDD. With the recursive predicate (`tc`) partitioned on the first argument notice how both the ERP and RRP terminate with a shuffle operator.



(a) TC (b) SG

Figure 5.10: PSN with SetRDD Physical Plans.

User-Defined Partitioning. In the plan in Figure 5.10(a) δ`tc` 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 second argument were instead made the default, the inefficiency with Figure 5.10(a) would be resolved but then other programs such as SG in Figure 5.10(b) would suffer (δ`sg` would require a shuffle prior to the join). Therefore, to support programs where the default partitioning will lead to inefficient execution, BigDatalog allows the user to define a recursive

predicate's partitioning via a configuration option. For example, by overriding the default partitioning and making `tc`'s second argument the partitioning strategy, the shuffle for δ`tc` before the join in Figure 5.10(a) will not be inserted to the plan and the plan shown in Figure 5.11 would be produced.

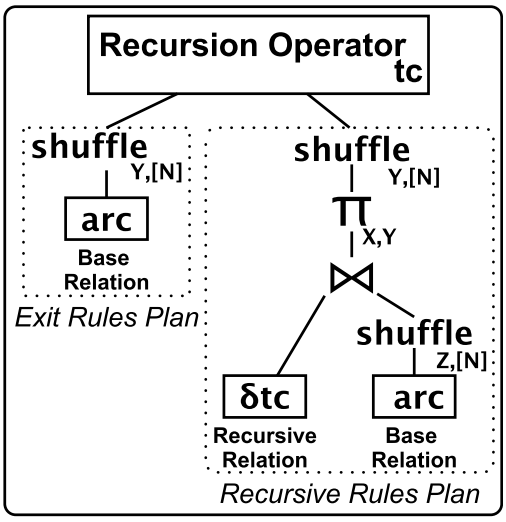


Figure 5.11: Plan for TC partitioned on 2nd argument.

Table 5.2 shows the results of TC evaluated with the plan in Figure 5.10(a) versus the same plan, but using the second argument as `tc`'s partitioning strategy. In fact, on all graphs from Table 5.6, the plan using the second argument matched or outperformed the other.

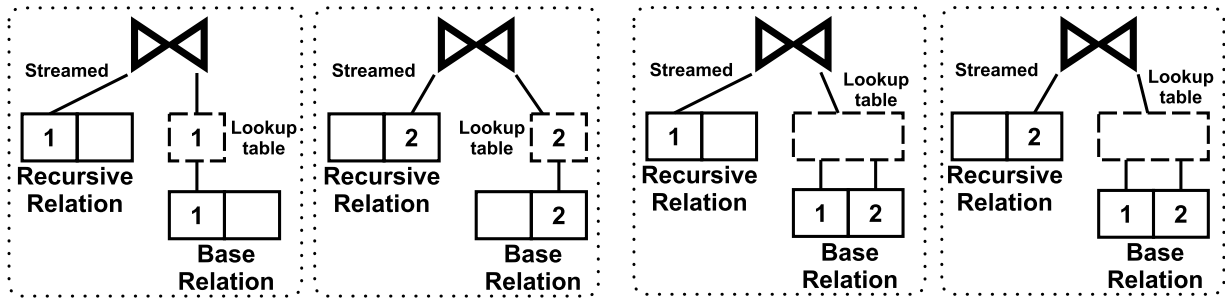
Table 5.2: Comparison of TC with Different Partitioning Strategies

Time (s)	`Tree17`	`Grid250`	`G10K`
1st Argument	41	370	20
2nd Argument	26	265	19

5.3.3 Join Optimizations for Linear Recursion

Input Caching. Since we use a static number of partitions and because non-recursive inputs do not change during evaluation, for a join implementing a linear recursion, the non-recursive join input can be cached. This can lead to significant performance improvements since input partitions

no longer have to be shuffled and loaded into lookup tables prior to a join each iteration. Table 5.3 shows the improved performance of caching. For TC on `Tree17`, shuffling and loading lookup tables for join each iteration is a significant part of the execution time, even though there are only 17 iterations. `Grid250` benefits because it requires 500 iterations and now only needs to load the lookup tables once.



(a) Shuffle Join (b) Broadcast Join

Figure 5.12: Linear Recursion Joins. Numbers 1 and 2 indicate partition ids of the respective dataset.

Broadcast Joins. Figure 5.12 depicts two approaches where the non-recursive input is a base relation and each input dataset has two partitions. For linear recursions, instead of joining partitions as described earlier and depicted in Figure 5.12(a), each partition of a recursive relation can be joined with an entire relation as depicted in Figure 5.12(b). We now refer to the former type of join as a *shuffle join* and the latter as a *broadcast join*[3]. In both scenarios, 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 every iteration. However, to execute a broadcast join the relation must fit in memory. With large memories currently available, we believe broadcast joins are acceptable for many DATALOG programs.

Figure 5.13 shows an RRP for Program 25 (SG) where the three-way join from the logical plan (Figure 5.5(b)) has been converted to a two-level broadcast join. In the event that a broadcast

[3]Broadcast joins are supported with Spark's broadcast variable infrastructure.

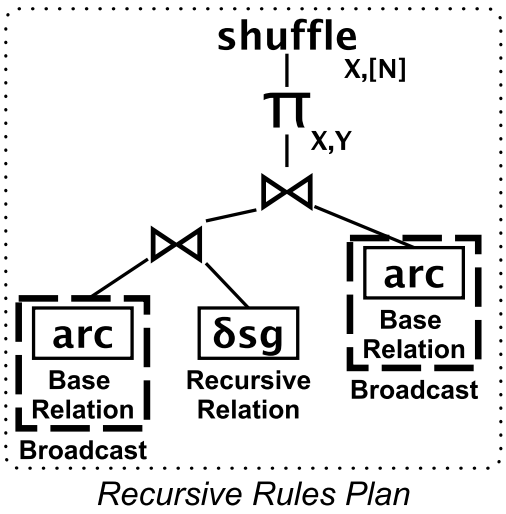


Figure 5.13: SG with Broadcast Joins.

relation is used multiple times in a plan, as in Figure 5.13, BigDatalog will broadcast it once and all broadcast join operators joining the relation will use it.[4]

Table 5.3 shows the results of using broadcast joins compared to shuffle joins on tree and grid graphs for TC and SG. SG greatly benefits on both graphs from using broadcast joins because three shuffles are eliminated from the plan and these graphs require minimal broadcast time. However, broadcast joins proved inefficient for `Tree17` for TC because the job to load the lookup table and broadcast it to workers takes half of the execution time. Nevertheless, broadcast join is the default join operator for linear recursion, but shuffle join can be selected via configuration setting.

Table 5.3: Join Optimizations for Linear Recursion

Time (s)	TC		SG	
	`Tree17`	`Grid250`	`Tree11`	`Grid250`
Shuffle join no caching	26	265	59	107
Shuffle join caching	17	196	56	81
Broadcast join	53	197	45	54

[4]We extend SparkSQL's broadcast join operator with support for sharing a broadcast relation between join operators.

5.3.4 Decomposable Programs

Previous research on parallel evaluation of DATALOG programs determined some programs are *decomposable* and thus evaluable in parallel without redundancy (a fact is only produced once) and without processor communication or synchronization [WS88]. Techniques for evaluating decomposable programs are appealing for BigDatalog because data-parallel systems like Spark can scale to utilize many machines and large numbers of cpu cores. Furthermore, mitigating the cost of synchronization and shuffling can lead to significant execution time speedup.

Program 1 (linear TC) is a decomposable program [WS88] and the following example demonstrates this. This example is based on the example in [WS88]. Using the facts from Figure 5.14, we make the evaluation of the exit rules result in two partitions. Figure 5.15 shows the derivations of the *r1* exit rule. One task creates a partition only with odd numbers for the first argument (left side of Figure 5.15) and a second task creates a partition only with even numbers for the first argument (right side of Figure 5.15). In Figure 5.15 we use an expression with a modulo operator in the exit rule to identify the partition number, however BigDatalog will actually use a shuffle operator to achieve this. Now entering the recursive part of the program, the work is partitioned for the evaluation of two independent TC evaluations, as long as `arc` is available for each evaluation. This is illustrated in Figure 5.16. One task produces the TC for vertex pairs starting at even numbered vertices, the other produces the TC for vertex pairs starting at odd numbered vertices. It should be clear how this approach can be extended to any number of tasks (i.e., use mod `N` where `N` is the number of tasks).

However, even if a program is decomposable, the system still needs to be able to produce physical plans to evaluate it as such. We consider a BigDatalog physical plan decomposable if RRP has no shuffle operators. The physical plan for Program 1 shown in Figure 5.10(a) has shuffle operators in RRP. Instead, BigDatalog can produce a decomposable physical plan for Program 1 by partitioning `tc` on the first argument and using a broadcast join. The partitioning strategy (first argument) divides the recursive relation so each partition can be evaluated independently and without shuffling, and the broadcast join allows each partition of the recursive relation to join with

Facts: $\texttt{arc}(0,2).\ \texttt{arc}(1,2).\ \texttt{arc}(2,3).\ \texttt{arc}(4,2).$

Figure 5.14: `arc` Facts for Example Decomposable Evaluation.

r1 Derivations	
Partition 0 (even)	**Partition 1 (odd)**
$\texttt{tc}(0,2) \leftarrow \texttt{arc}(0,2), 0 \bmod 2 = 0.$	$\texttt{FAIL} \leftarrow \texttt{arc}(0,2), 0 \bmod 2 \neq 1.$
$\texttt{FAIL} \leftarrow \texttt{arc}(1,2), 1 \bmod 2 \neq 0.$	$\texttt{tc}(1,2) \leftarrow \texttt{arc}(1,2), 1 \bmod 2 = 1.$
$\texttt{tc}(2,3) \leftarrow \texttt{arc}(2,3), 2 \bmod 2 = 0.$	$\texttt{FAIL} \leftarrow \texttt{arc}(2,3), 2 \bmod 2 \neq 1.$
$\texttt{tc}(4,2) \leftarrow \texttt{arc}(4,2), 4 \bmod 2 = 0.$	$\texttt{FAIL} \leftarrow \texttt{arc}(4,2), 4 \bmod 2 \neq 1.$
$\delta\texttt{tc}_0 : \{\texttt{tc}(0,2).\ \texttt{tc}(2,3).\ \texttt{tc}(4,2).\}$	$\delta\texttt{tc}_1 : \{\texttt{tc}(1,2).\}$

Figure 5.15: Example *r1* Derivations for Decomposable TC Evaluation.

	r2 Successful Derivations	
	Even Partition	**Odd Partition**
1st Iteration	$\texttt{tc}(0,3) \leftarrow \texttt{tc}(0,2), \texttt{arc}(2,3).$ $\texttt{tc}(4,3) \leftarrow \texttt{tc}(4,2), \texttt{arc}(2,3).$	$\texttt{tc}(1,3) \leftarrow \texttt{tc}(1,2), \texttt{arc}(2,3).$
	$\delta\texttt{tc}_0 : \{\texttt{tc}(0,3).\ \texttt{tc}(4,3).\}$	$\delta\texttt{tc}_1 : \{\texttt{tc}(1,3).\}$
2nd Iteration	NONE	NONE
	$\delta\texttt{tc}_0 : \{\varnothing\}$	$\delta\texttt{tc}_1 : \{\varnothing\}$
Fixpoint Reached		

Figure 5.16: Example *r2* Derivations for Decomposable TC Evaluation.

the *entire* `arc` base relation. Figure 5.17 is the decomposable physical plan for Program 1. Since we do not pre-partition base relations, the ERP has a shuffle operator to repartition the `arc` base relation into `N` partitions by `arc`'s first argument `X`.

Figure 5.18 depicts the RDD lineage graph produced when evaluating the physical plan shown in Figure 5.17 up through two iterations of PSN. Mapping the evaluation of Figure 5.17 with PSN into RDDs is straightforward, however there are some important points to take note of. Firstly, the shuffle in the ERP is the only shuffle in this RDD lineage graph. Secondly, the broadcast `arc`

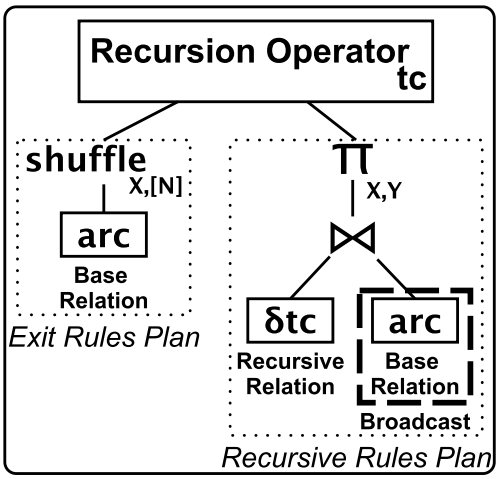


Figure 5.17: Decomposable TC Plan.

relation is copied to each worker where it is cached and reused each iteration.

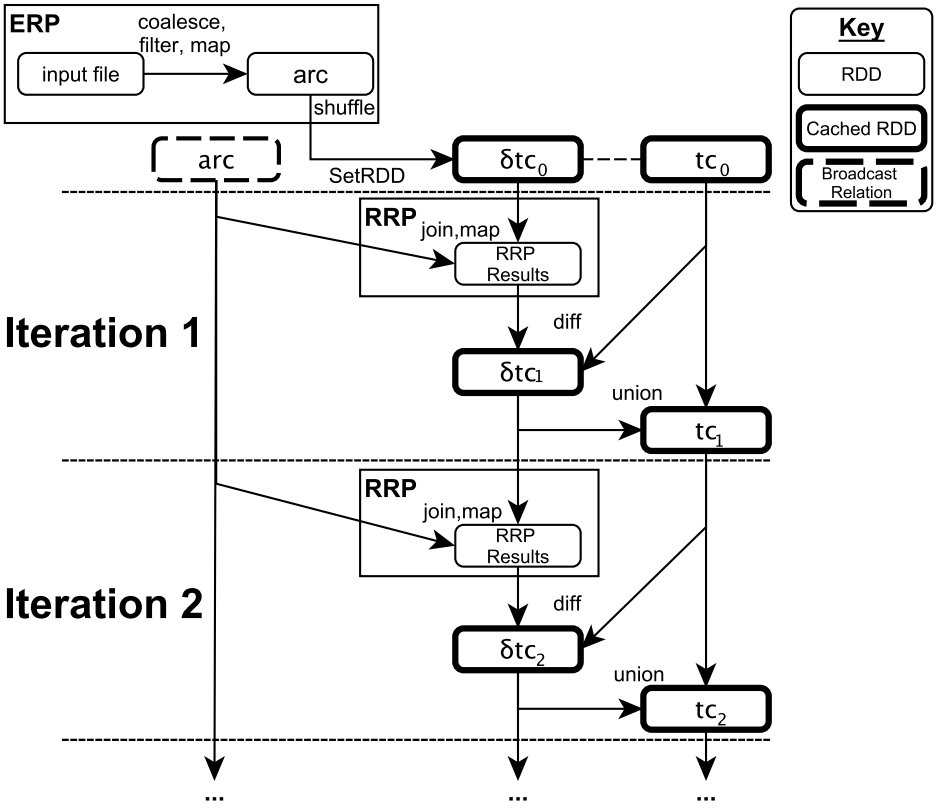


Figure 5.18: RDD Lineage Graph for Decomposable TC Physical Plan.

Table 5.4 displays the execution times using the shuffle join plan and the decomposable plan in Figure 5.17. With the exception of `Tree17`, the decomposable plan greatly outperforms the plan

using shuffle joins.

Table 5.4: Comparison of Shuffle vs. Decomposable TC Plans

Time (s)	Tree17	Grid250	G10K	G10K-0.01	G20K
Shuffle	26	265	19	121	101
Decomposable	49	55	7	22	19

Although linear TC is decomposable, not all linear recursions are. Unary (single argument) predicates such as `reach` in Program 27 are not decomposable. The Reachability (REACH) program identifies all nodes reachable from the given source node. *r1* initializes the recursion from 0. Then, in *r2* previously computed `reach` facts are joined to `arc` to find new vertices reachable from 0.

Program 27. *Reachability*

$$r1.\texttt{reach}(\texttt{Y}) \leftarrow \texttt{Y} = 0.$$
$$r2.\texttt{reach}(\texttt{Y}) \leftarrow \texttt{reach}(\texttt{X}),\ \texttt{arc}(\texttt{X}, \texttt{Y}).$$

Identifying Decomposable Programs. BigDatalog identifies decomposable programs via syntactic analysis of program rules using techniques presented in the *generalized pivoting* work [SL91]. The authors of [SL91] show that the existence of a *generalized pivot set* (GPS) for a program is a sufficient condition for decomposablility and present techniques to identify GPS in arbitrary DATALOG programs. We have implemented the techniques described in [SL91] to determine the GPS for BigDatalog programs. When a BigDatalog program is submitted to the compiler, the compiler will apply the generalized pivoting solver to determine if the program's recursive predicates have GPS. If they indeed do, we now have a partitioning strategy and, in conjunction with broadcast joins, we can efficiently evaluate the program with these settings. For example, Program 1 has a GPS which says to partition the `tc` predicate on its first argument.

Note that this technique is enabled by using DATALOG which allows BigDatalog to analyze the program at the logical level. The Spark API is unable to provide this type of support alone because programs are written in terms of physical operations.

5.3.5 Job Optimizations

Lineage. Since RDDs produced during an iteration are input for the next iteration, RDD lineage can grow long for recursive programs. Since lineage is inspected frequently during execution, for long running recursions we found this can result in a stack overflow. The standard solution is to checkpoint the RDD which clears the lineage after being written to disk, however a disk write is not always acceptable. To address this problem we implement a technique for cached RDDs that will clear lineage, but does not require checkpointing.[5] We sacrifice some degree of fault tolerance in favor of execution time performance, although this technique can still utilize cache replication and disk backup. Otherwise, we leverage the standard fault tolerance mechanisms provided by Spark.

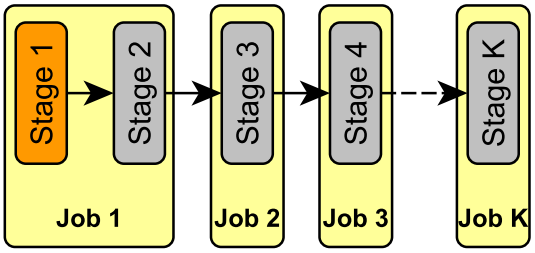


Figure 5.19: Multi-Job scheduling of Program 1 (TC). ShuffleMapStages are orange. ResultStages, which produce output for a job, are gray. Job 0 is a broadcast of the `arc` base relation.

Scheduler-Aware Recursion. With PSN as shown in Algorithm 5, the scheduler is unaware that subsequent iterations could be required and therefore is unable to optimize recursive execution. We now refer to this approach as *Multi-Job PSN*, which is depicted in Figure 5.19 for evaluating Program 1. To address the inefficiency with Multi-Job PSN, we investigate pushing the recursion into the scheduler so recursive queries are supported as *Single-Job PSN*. We extend the Spark scheduler to use a special type of stage for recursion (*FixpointStage*) and support a *fixpoint job*, which is different from normal jobs in that 1) each iteration, the scheduler evaluates a new RDD over the previous iteration's results and 2) the scheduler will issue iterations until evaluation of the RDD results in an empty RDD indicating a fixpoint has been reached. Figure 5.20 depicts

[5] A solution to this problem in Spark 1.5.0 is to use *Local Checkpointing*, i.e., checkpointing on the local worker.

evaluating Program 1 in this manner. Lastly, checkpointing is applied at the end of a job, so previously each iteration of Multi-Job PSN could be checkpointed. To support checkpointing an iteration in Single-Job PSN, checkpointing is also pushed into the scheduler.

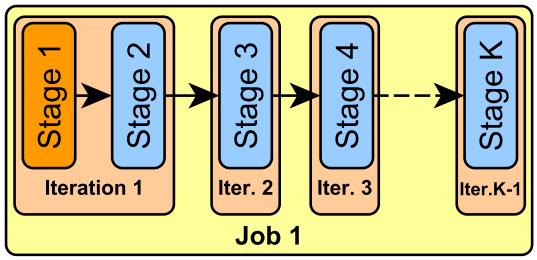


Figure 5.20: Single-Job scheduling of Program 1 (TC). ShuffleMapStages are orange. ResultStages, which produce output for a job, are gray. FixpointResultStages are blue. Job 0 is a broadcast of the `arc` base relation.

Optimizing Single-Job PSN. With Single-Job PSN, the scheduler is now aware that multiple iterations could be required. If a program is partitioned such that it does not require shuffling in the recursion, the scheduler will not create stages with shuffle operators. When the scheduler detects this situation, it configures the stage's tasks to iterate on workers and execute the same RDD until a fixpoint is reached. To support reusing the same RDD, the RDD partitions in the local cache from the previous iteration are overwritten with the RDD partitions produced during the current iteration. We call this *Single-Job PSN Reuse*. This approach eliminates the cost of scheduling and task creation for subsequent iterations. Figure 5.21 depicts this approach for Program 1 (TC) evaluated with the plan from Figure 5.17.

Table 5.5 displays results of the execution times of TC and SG using the Multi-Job PSN, Single-Job PSN and Single-Job PSN Reuse. For datasets that require many iterations, such as `Grid250`, the performance improvement is substantial.

Note that this scheduler optimization is used on decomposable programs. Being able to identify a decomposable program (i.e., generalized pivoting) is independent from this optimization and thus this can be used in general to evaluate a decomposable plan, not just by BigDatalog.

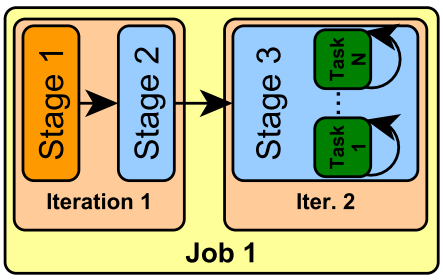


Figure 5.21: Single-Job Reuse scheduling of Program 1 (TC). ShuffleMapStages are orange. ResultStages, which produce output for a job, are gray. FixpointResultStages are blue. Job 0 is a broadcast of the `arc` base relation.

Table 5.5: Comparison of PSN Job Strategies

Time (s)	TC		SG	
	`Tree17`	`Grid250`	`Tree11`	`Grid250`
Multi-Job PSN	51	111	53	75
Single-Job PSN	49	55	53	53
Single-Job PSN Reuse	45	26	N/A	N/A

5.4 Aggregates

Traditional SQL aggregates (`min`, `max`, `sum`, `count`, `avg`) are non-monotonic and thus cannot be used in recursion. Researchers have recently proposed aggregates that are monotonic w.r.t. set containment, the same monotonicity used by standard Datalog, meaning these aggregates can be used in recursive rules and evaluated using techniques such as SN and magic sets [MSZ13a, MSZ13b]. A sequential version of these aggregates was presented in Chapter 4. In this chapter we present a distributed version of the aggregates.

BigDatalog supports the same four monotonic aggregates as *DeALS* - `mmin`, `mmax`, `mcount`, `msum`. The declarative semantics allows the aggregates inside the recursion so long as monotonicity w.r.t. set containment is maintained. Therefore, during evaluation the monotonic aggregates can produce new higher (`mmax`, `mcount`, `msum`) or lower (`mmin`) values with each input fact and thus an outer non-monotonic aggregate (`min` or `max`) is necessary to produce only the final value.

An example of this can be seen in Program 28, the single-source shortest paths program (SSSP). Note, BigDatalog uses aggregates functions in rule heads with the non-aggregate arguments as the grouping arguments.

Program 28. *Single-Source Shortest Paths*

$$r1.\ \texttt{sssp2}(\texttt{Y}, \texttt{mmin}\langle\texttt{D}\rangle) \leftarrow \texttt{Y} = 1, \texttt{D} = 0.$$

$$r2.\ \texttt{sssp2}(\texttt{Y}, \texttt{mmin}\langle\texttt{D}\rangle) \leftarrow \texttt{sssp2}(\texttt{X}, \texttt{D1}), \texttt{arc}(\texttt{X}, \texttt{Y}, \texttt{D2}), \texttt{D} = \texttt{D1} + \texttt{D2}.$$

$$r3.\ \texttt{sssp}(\texttt{X}, \texttt{min}\langle\texttt{D}\rangle) \leftarrow \texttt{sssp2}(\texttt{X}, \texttt{D}).$$

The SSSP program computes the length of the shortest path from a source vertex to all vertices it is connected to. This program uses a `mmin` monotonic aggregate. Here the `arc` predicate in *r2* denotes edges of the graph (`X`, `Y`) with edge cost `D2`. *r1* seeds the recursion with starting vertex 1. Then, *r2* will recursively produce all new minimum cost paths to a node `Y` though node `X`. Lastly, *r3* produces only the minimum cost path for each node `X`, however in our actual implementation, we do not have to evaluate *r3* since at the completion of the recursion, `sssp2`'s relation will contain the shortest path from 1 to each vertex.

Evaluation and Implementation. Programs with monotonic aggregates in recursive rules are evaluated with an aggregate version of PSN we call *Parallel Semi-Naïve - Aggregate* (PSN-A). Compared with PSN, PSN-A is a simpler framework. Since new facts are only produced when a greater (`mmax`, `mcount`, `msum`) or lesser (`mmin`) value than the previous value for the (aggregate) group is produced, de-duplication is unnecessary. Furthermore, the union operation performed in PSN is unnecessary in PSN-A because new results are added to the aggregate relation during aggregate evaluation. We implement PSN-A in an aggregate version of an RO. Also, we use a specialized RDD called an *AggregateSetRDD*, in which each partition is a key/value map where each entry represents a unique group and its current value. We use B+trees as the data structures for the maps and cache the AggregateRDD to avoid the expense of reloading the maps each iteration. Additionally, since the aggregates functions are monotonic, as with SetRDD's `union` operation, AggregateSetRDD is mutable under aggregate evaluation. Furthermore, an AggregateSetRDD will reference the same maps as its creator. Should a task fail during evaluation, any changes to the

RDD partition will not result in incorrect results since a value can only be updated if it is higher (mmax, mcount, msum) or lower (mmin) than previously computed values.

For efficient distributed evaluation, monotonic aggregates require a partitioning strategy such that a group exists in only one partition. If a group is allowed in multiple partitions, a task could produce a value even though a better value exists. If the input partitioning does not match the aggregate's grouping arguments, the aggregate's input is shuffled. At the end of an iteration, the aggregate will produce a set containing the last entry made by any group updated during the iteration. This set is then used in the next iteration (i.e., δ).

Program 29. *Connected Components*

$$r1.\ \texttt{cc2}(\texttt{X}, \texttt{mmin}\langle\texttt{X}\rangle) \leftarrow \texttt{arc}(\texttt{X}, \_).$$
$$r2.\ \texttt{cc2}(\texttt{Y}, \texttt{mmin}\langle\texttt{Z}\rangle) \leftarrow \texttt{cc2}(\texttt{X}, \texttt{Z}), \texttt{arc}(\texttt{X}, \texttt{Y}).$$
$$r3.\ \texttt{cc}(\texttt{X}, \texttt{min}\langle\texttt{Y}\rangle) \leftarrow \texttt{cc2}(\texttt{X}, \texttt{Y}).$$

The connected components (CC) program is for identifying the connected components of a graph. This program works by initially assigning the node's id to itself (*r1*) and then propagating a new lower node id for any edge the node is connected to. *r3* is necessary to select only the minimum node id Y for each X found in cc2.

Lastly, in addition to the aggregate programs presented in this chapter, BigDatalog also supports programs such as Programs 4 and 5 presented in Chapter 4.

5.5 Experiments

In this section, we present experimental results comparing BigDatalog against other large-scale DATALOG systems. Additionally, we perform experiments to understand how BigDatalog scales as either cluster size or dataset size increases.

5.5.1 Experimental Setup

Our experiments are conducted on a 16 node cluster. Each node runs Ubuntu 14.04 LTS and has an Intel i7-4770 CPU (3.40GHz, 4 core/8 thread), 32GB memory and a 1 TB 7200 RPM hard drive. Nodes of the cluster are connected with 1Gbit network. BigDatalog is implemented using Spark 1.4.0 and Hadoop 1.0.4.

Table 5.6: TC and SG Synthetic Test 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

Datasets. Table 5.6 displays the synthetic graphs used for TC and SG experiments. We use these graphs to understand how BigDatalog evaluates TC and SG on graphs exhibiting specific structural properties. `Tree11` and `Tree17` are trees of height 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 G*n*-*p* graphs are *n*-vertex random graphs (Erdős-Rényi model) generated by randomly connecting vertices so that each pair is connected with probability *p*. G*n*-*p* graph names omitting *p* use default probability 0.001. Graphs in Table 5.6 with an asterisk were taken from [YZ14].

Experiments on REACH (Program 27), CC (Program 29) and SSSP (Program 28) use the RMAT graphs `RMAT-`n for $n \in \{$1M, 2M, 4M, 8M, 16M, 32M, 64M, 128M$\}$ generated by the RMAT graph generator [gtg15] with parameters $(a, b, c) = (0.45, 0.25, 0.15)$. `RMAT-`$n$ has n vertices and $10n$ directed edges with uniform integer weights range from $[0, 100)$.

5.5.2 Datalog Systems Comparison

In this section, we report experimental results comparing BigDatalog against other large-scale DATALOG systems. The purpose of this comparison is to show that with the enhancements and optimizations proposed in this dissertation Spark is indeed an efficient runtime for DATALOG and recursive applications. Next, we present the systems we compared with.

Myria has been extended to support asynchronous DATALOG evaluation [WBH15]. Its runtime is a pipelined, parallel, distributed execution engine that evaluates a graph of operators which is broken down into fragments, where each fragment is executed in a separate thread on worker nodes. Datasets are sharded and stored in PostgreSQL instances at worker nodes and read entirely into memory during evaluation. Myria uses hash partitioning.

Distributed SociaLite is a DATALOG language implementation for social network analysis [SPS13]. SociaLite programs are code generated and evaluated with a parallel engine on each worker. SociaLite supports range and hash partitioning and uses message passing to transfer data between workers.

For our experiments we also compare with native Spark and use two approaches. For TC and SG, we use optimized Semi-Naïve programs written in the Spark API. These programs attempt to reduce any unnecessary shuffling. For the experiments on REACH, CC and SSSP, we use programs for GraphX, Spark's graph processing module that implements Pregel [MAB10] on top of Spark. [GXD14] showed GraphX outperforms native Spark on these types of programs, which we validated in our experimental environment.

For each system, one machine was dedicated as the master and each of the 15 worker nodes was allowed 30 GB RAM and 8 CPU cores (120 total cores). Myria was configured with one instance of Myria and PostgreSQL per node, since each node has one disk, which was confirmed as appropriate by the author of [WBH15]. For Spark programs and BigDatalog, we evaluate with one partition per available CPU core. BigDatalog uses Single-Job PSN with SetRDD.

5.5.2.1 TC and SG Experiments

We perform experiments comparing the execution time of BigDatalog for TC and SG programs with Myria, SociaLite and handwritten Spark programs on graphs listed in Table 5.6. Although these graphs appear small in terms of number of vertices and edges, TC and SG are capable of producing result sets many orders of magnitude larger than the input dataset. BigDatalog is the only system that finishes the evaluation for TC and SG on all graphs in Table 5.6, except SG on `Tree17` since the size of the result is larger than the total disk space of the cluster. Figure 5.22 and Figure 5.23 display the evaluation times for TC and SG, respectively, for all four systems. Note, larger graphs are used in experiments later in this section, albeit they are only evaluated using BigDatalog, and we do not display those results here.

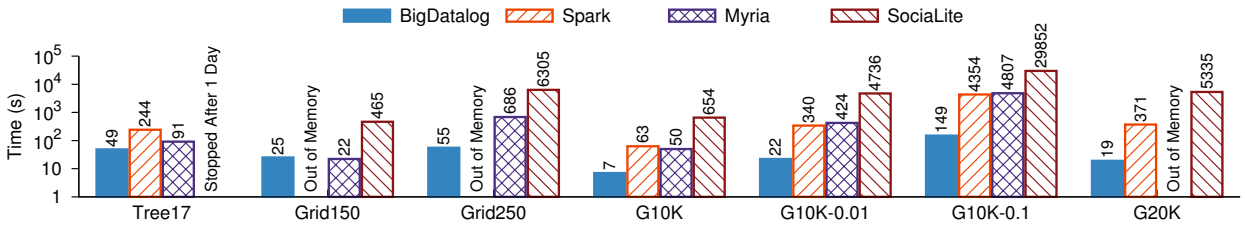


Figure 5.22: TC System Comparison.

Aside from `Grid150`, BigDatalog has the best execution time on all graphs for TC, and on four of the remaining six graphs outperforms the other systems by an order of magnitude. BigDatalog uses the decomposed plan from Figure 5.17 for TC, which only performs an initial shuffle of the dataset. This design shows to be effective for evaluating TC on these graphs. However, if the evaluation requires many iterations while very little work is performed in each iteration, the overhead of scheduling takes a significant portion of execution time, and thus BigDatalog performs slightly slower compared to Myria on `Grid150`. If BigDatalog instead evaluates `Grid150` with Single-Job PSN Reuse (Section 5.3.5), the execution time drops to thirteen seconds. The handwritten Spark programs are also affected by the overhead of scheduling on `Grid150` and `Grid250`, which require 300 and 500 iterations, respectively, but also suffer memory utilization issues related to dataset caching and run out of memory. For the remaining five graphs, the native Spark programs are slower compared with BigDatalog due to the overhead of shuffling. The same amount

of data is also transmitted via shuffling or message passing for both Myria and SociaLite. But their performance is less stable compared with Spark, e.g., Myria runs out of memory on `G20K` and SociaLite is always more than 10X slower, since the implementation of their communication subsystem is less robust compared to Spark's.

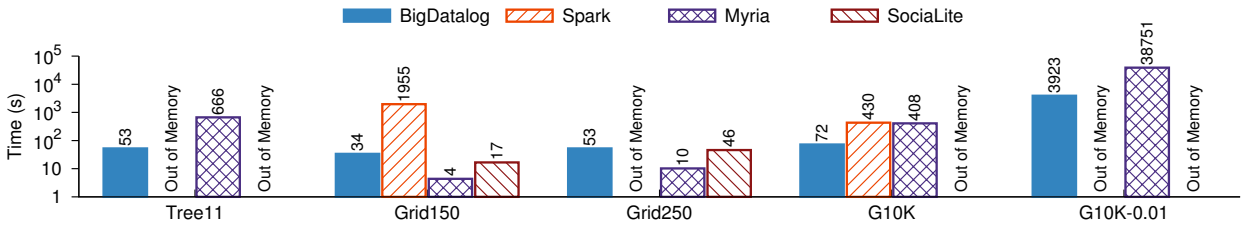


Figure 5.23: SG System Comparison.

SG is a "harder" program than TC, because shuffling is required and two joins are evaluated each iteration. BigDatalog uses the plan with broadcast joins whose RRP is shown in Figure 5.13. BigDatalog performs the best on graphs other than `Grid150` and `Grid250`. BigDatalog is slower than Myria and SociaLite on both graphs partly due to the overhead of scheduling, but the evaluation of these two graphs produces the smallest result sets as well as the least amount of intermediate results. Unlike with TC on `Grid150`, the handwritten SG Spark program can evaluate `Grid150` because there is half as many iterations with SG than with TC and therefore less caching is necessary. However, SG with native Spark is over 50X slower than BigDatalog on `Grid150` since BigDatalog only requires a single shuffle per iteration due to the broadcast joins, whereas the Spark program has to shuffle between two nested-loop joins. Additionally, the handwritten SG Spark program runs out of memory on three graphs due to caching. Lastly, BigDatalog is faster than Myria and SociaLite on `Tree11`, `G10K` and `G10K-0.01` partly due to the more robust shuffling implementation Spark provides.

Map-side Distinct. Recall the RRP for SG in Figure 5.13. Within a task, two joins will be evaluated and the projected output will be input to a shuffle. The joins can generate a massive amount of intermediate duplicate results, however de-duplication occurs after the shuffle (in PSN). Therefore, to prevent a large amount of data being written to disk for the shuffle, we place a `distinct` operator into the plan immediately before the shuffle, much like a map-side combiner.

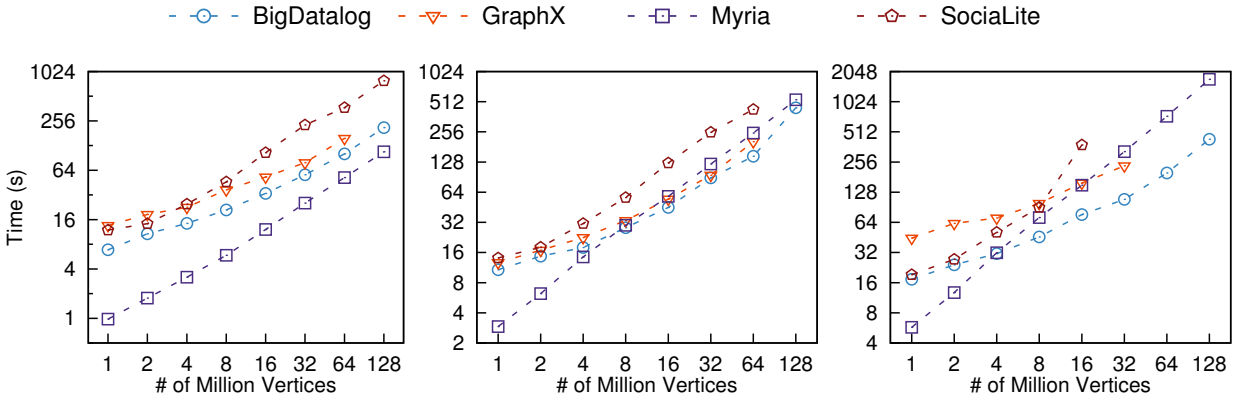
This has no effect on program correctness because of DATALOG's set semantics. This has a minor affect on execution time performance on the smaller graphs (i.e., `Grid150`) but allows BigDatalog to better support larger graphs.

5.5.2.2 REACH, CC and SSSP Experiments

We perform experiments comparing the execution time of BigDatalog for REACH, CC and SSSP programs with Myria, SociaLite and GraphX programs on RMAT graphs. These results also help us understand how BigDatalog scales on different programs as the graph sizes increase compared to other systems. Figure 5.24 shows the experimental results in which the x-axis represents test graphs from `RMAT`-1M to `RMAT`-128M. For each system, we report the total time of evaluation starting from loading the data from persistent storage, i.e., from PostgreSQL for Myria and from HDFS for the remaining three systems, until the evaluation completes. For Figure 5.24(a) and Figure 5.24(c) each point represents the average time to evaluate the program on the test graph over a set of randomly selected vertices. A point is not reported in a figure if a system runs out of memory for the experiment.

We use three graph queries in this comparison. REACH (Program 27) finds all vertices connected by some path to a given source vertex with a simple linear recursion; CC (Program 29) uses a label propagation approach to determining the lowest vertex id a vertex is connected to, thus establishing membership in a component, and this program uses a monotonic `mmin` aggregate in the recursion. SSSP (Program 28) also uses a `mmin` aggregate and a linear recursion. All three programs require shuffling during evaluation. Let n, m and d be the number of vertices, number of edges, and diameter of a graph, respectively. The number of intermediate results produced during evaluation is $O(m)$, $O(dm)$ and $O(nm)$ for REACH, CC and SSSP, respectively. Figure 5.24 displays the results for the three programs by increasing order of the amount of communication as $1 \leq d \leq n$.

Figure 5.24(a) shows that Myria performs the best on all test instances for REACH. BigDatalog narrows the execution time gap with Myria as the size of the graph increases due to the increased



(a) REACH (b) CC (c) SSSP

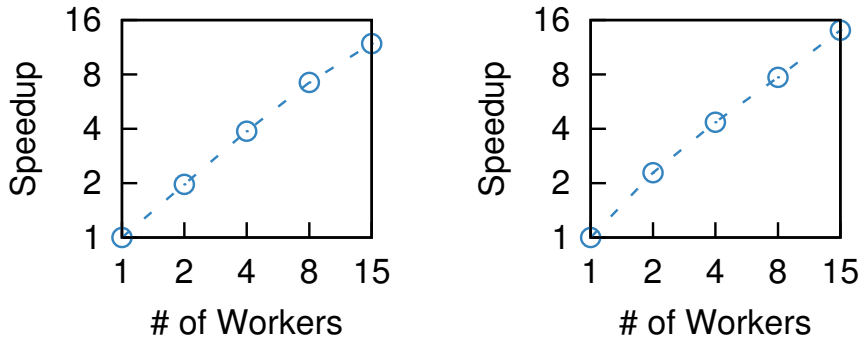
Figure 5.24: System Scaling-up Comparison on RMAT Graphs.

amount of communication. As the amount of communication further increases in CC and SSSP, BigDatalog outperforms Myria starting from RMAT-8M for CC and RMAT-4M for SSSP as shown in Figure 5.24(b) and Figure 5.24(c), respectively. SociaLite's performance on all three experiments is impacted because of loading and initialization of base relations and SociaLite takes significantly longer at this task than the other systems. As explained to us by an author of [SPS13], the cause of this is that SociaLite is implemented for a much faster network connection than we have available. However, none of the other systems suffer from these same inefficiencies. In general, as with TC and SG, BigDatalog benefits from Spark's efficient shuffling implementation compared to the less efficient message passing implementations used by Myria and SociaLite. Finally, BigDatalog always outperforms GraphX on these experiments. BigDatalog has less scheduling overhead by using a single job versus a job for each iteration in GraphX. Additionally, BigDatalog uses more efficient data structures, *w.r.t.* size in memory, compared to GraphX's vertex and edge RDDs, which each require maintaining more data structures than the SetRDD/AggregateSetRDD design used in BigDatalog.

5.5.3 Additional Scaling Experiments

We have shown how BigDatalog scales for REACH, CC and SSSP on RMAT graphs. In this section, we report additional experimental results of how BigDatalog scales over different cluster

and dataset sizes for TC and SG.



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

Figure 5.25: Scaling-out Cluster Size.

Scaling-out. We investigate how BigDatalog scales over different cluster sizes. We use the largest G*n-p* graphs that could be evaluated on all cluster sizes. Figure 5.25(a) shows the speedup for TC on G20K as the number of workers increases from one to 15 (all with one master) *w.r.t.* using only one worker, and Figure 5.25(b) shows the same experiment run for SG with G10K. Both figures show a linear speedup. Using 15 workers, TC and SG are 12X and 14X faster, respectively, than with a single worker.

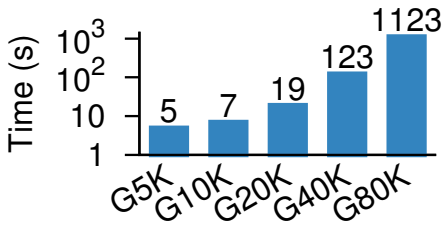


Figure 5.26: Scaling-up TC on Random Graphs.

Scaling-up. We use the full cluster to see how BigDatalog scales over random graphs from Table 5.6 of increasing sizes for TC and SG. Broadcasting the `arc` relation requires between one second for G5K to twelve seconds for G80K. In this experiment, with each successively larger graph size (e.g.,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 focus 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 5.7.

Table 5.7: TC Scaling Experiments Result Details

Graph	Time - broadcast[a] (s)	TC	Generated Facts	Generated Facts / TC	Generated Facts / Sec.
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

[a] - execution time not including the time to broadcast arc.

Among the items displayed in Table 5.7 is the execution time minus the time to broadcast arc, which is the total time the program required to actually evaluate TC. Table 5.7 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 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.

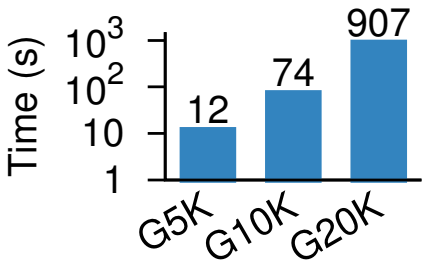


Figure 5.27: Scaling-up SG on Random Graphs.

Figure 5.27 displays the scaling up experimental results for SG. Since we double the size of the graph starting from G5K, random graphs larger than G20K exceed our cluster's resources. For these experiments, the broadcast of the arc relation was a less significant part of the overall execution time requiring between one second for G5K and two seconds for G20K.

Table 5.8: SG Scaling Experiments Result Details

Graph	Time - broadcast[a] (s)	SG	Generated Facts	Generated Facts / SG	Generated Facts / Sec.
G5K	11	24,611,547	612,891,161	24.90	55,717,378
G10K	73	100,000,000	10,037,915,957	100.38	137,505,698
G20K	905	400,000,000	159,342,570,063	398.36	176,069,138

Table 5.8 displays the same details as Table 5.7 but for SG. Table 5.8 displays the execution time-minus the broadcast time of arc, the result set size, the number of 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.

Lastly, to evaluate these graphs for SG, we used the map-side distinct optimization described in Section 5.5.2.1. Table 5.9 shows the impact of using this optimization. The trade off here is that the distinct operator will filter out duplicates prior to being shuffled and de-duplicated in PSN. The trade off here is the memory required for the distinct operator versus the additional disk space required to shuffle the un-deduplicated results. From Table 5.9 we see the effect on execution time is minimal on the smallest graph tested (one second). However as the graph size increases and the number of facts shuffled greatly increases, the effect on execution time is substantial.

Table 5.9: Impact of Map-side Distinct on SG Scaling Experiments

	Map-side Distinct = true		Map-side Distinct = false		
Graph	**Time (s)**	**Facts Shuffled**	**Time (s)**	**Facts Shuffled**	**Ratio**
G5K	12	226,350,156	13	612,891,161	2.71
G10K	74	1,982,954,558	121	10,037,915,957	5.06
G20K	907	9,894,980,670	6645	159,342,570,063	16.10

5.6 Related Works

We first review works on parallel DATALOG evaluation and then discuss related systems for large-scale data analysis.

Parallel Datalog Evaluation. We have previously discussed the contributions of [WS88] with decomposable programs and the work of generalized pivoting [SL91]. Early work on parallelization of bottom-up evaluation of DATALOG programs was largely of a theoretical nature for instance [Van86] proposed a message passing framework for parallel evaluation of logic programs. Techniques to partition program evaluation efficiently among processors [WO90], the tradeoff between redundant evaluation and communication [GST90, GST92] and classifying how certain types of DATALOG programs can be evaluated [CW89] were also studied. A parallel Semi-Naïve fixpoint has been proposed for a message passing design [WO90] 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 of the database at each worker and uses shuffle operators in place of processor communication.

[BBC12] showed how to use XY-stratified DATALOG to support computational models for large-scale machine learning, although no full DATALOG language implementation on a large-scale system was provided. Recent theoretical work on recursive query evaluation showed a version of non-linear transitive closure that is more efficient than linear versions for distributed settings [AU12], however this work never addresses the problem of how to convert arbitrary programs to this desirable form. The *Bloom*^{L} [CMA12] distributed programming language uses various

monotonic lattices to identify program elements not requiring coordination.

Systems for Large Scale Data Analysis. We have previously discussed [AXL15, SPS13, WBH15, ZCD12] in earlier sections. DryadLINQ [YIF08] provides a high level imperative/declarative language that relies on control flow in its host language to support iteration. Extended MapReduce system designs providing API support include Haloop [BHB12] (loop-aware scheduling and caching), PrIter [ZGG11] (prioritized execution of high priority nodes), and Twister [ELZ10] (in-memory iterative MapReduce with cachable tasks). Incremental iterations were integrated into Stratosphere (now Apache Flink ([apa15a]) to support iterative algorithms with sparse computational dependencies [ETK12]. ScalOps [WCR11], a Scala DSL designed for machine learning, supports a loop construct to include iteration in a recursive query plan executed on Hyracks [BCG11], a distributed dataflow engine. Naiad [MMI13] uses a time-based dataflow computational model with a vertex-based programming model to support iterative workflows and incremental updates. The SCOPE [ZBW12] system includes a SQL-like language that provides map and reduce-style operators and allows user defined aggregates (UDA) to be recursive. REX [MIG12], a distributed parallel engine, provides a SQL-based language and supports incremental updates and recursive queries via a UDA framework. [OKH13] presents an extension of SQL that includes constructs to specify incrementally computable views for more efficient MapReduce execution. Lastly, systems for both parallel [KBG12, LGK10] and distributed [gir15, GLG12, LBG12, MAB10, TBC13] graph analytics have been proposed however, these systems only support graph workloads and require programming against a low-level API.

5.7 Summary of BigDatalog

In this chapter, we presented BigDatalog, a DATALOG language implementation on Apache Spark. Using our system Spark programmers can now benefit from using a declarative, recursive language to implement their distributed algorithms, while maintaining the efficiency of highly optimized programs. On our large test graph instances BigDatalog outperforms other state-of-the-art DATALOG systems on the majority of our tests. Moreover, our experimental results confirmed that

among Spark-based systems BigDatalog outperforms both GraphX and native Spark for recursive queries.

Addressing our Challenges. We addressed the challenges for using Spark as a DATALOG runtime as outlined in Section 5.1.1 as follows: now with BigDatalog, recursive queries are compiled and optimized for efficient evaluation on Spark, which was verified with our experimental results in Section 5.5 (Challenge 1). We implemented BigDatalog to identify and produce physical plans for efficiently evaluating decomposable programs. In addition, we propose a new type of job for recursive programs to allow the scheduler greater control over iterations (Challenge 2). Lastly, we propose the SetRDD and AggregateSetRDD, specialized RDDs that utilize DATALOG semantics to support memory-efficient recursive evaluation in Spark (Challenge 3).

CHAPTER 6

Conclusion and Future Work

In this dissertation, we have presented our contributions towards the language challenges for advanced analytics as outlined in our introduction.

We have developed $DeALS$, a DATALOG system for analytics. We have designed and implemented a set of monotonic aggregates in $DeALS$ that can be used in recursion. These aggregates expand the range of programs $DeALS$ can efficiently support, which includes many graph queries and graph analytics that were otherwise difficult to express or inefficient to evaluate using previous constructs. We have demonstrated the efficiency of $DeALS$ through an experimental comparison with other DATALOG systems and found that $DeALS$ shows it is possible to provide a general DATALOG system capable of supporting a wide range of programs, that also provides superior performance.

We have designed and implemented BigDatalog – $DeALS$ on Apache Spark. We presented compiler and optimizer techniques to efficiently evaluate BigDatalog programs on Spark. We proposed job scheduling optimizations and show how certain DATALOG programs can be evaluated without communication during recursion in Spark. We conducted an experimental evaluation and compared BigDatalog with other large-scale DATALOG systems on both classical recursive queries, such as transitive closure, as well as aggregate queries. Compared with the other systems, BigDatalog outperformed them on many types of programs, including the classical recursive queries on larger graphs. Compared with native Spark programs, BigDatalog exhibited at least an order of magnitude improvement for almost every graph tested, and also outperformed Spark's GraphX module. Lastly, the experiments verified that our approach in general is quite effective in supporting DATALOG-based analytics on Spark.

In the course of conducting the research for this dissertation, several opportunities for exciting new research has been identified. In the area of distributed DATALOG evaluation, one first direction is to extend BigDatalog to support XY-DATALOG and realize the vision of [BBC12] to use DATALOG to support complex machine learning analytics such as logistic regression over a massively parallel system. Another direction is to investigate system extensions for provenance and fault tolerance enabled by efficient monotonic DATALOG constructs. Lastly, an exciting area for future research that builds on this dissertation's work on supporting declarative languages over multiple runtimes is to provide (i) a rule-based or cost-based optimizer to determine the appropriate runtime (e.g., sequential, distributed, etc.) and (ii) an accompanying scheduler to provide an optimized evaluation for $DeAL$ users.

REFERENCES

[ABH10] Azza Abouzied, Kamil Bajda-Pawlikowski, Jiewen Huang, Daniel J. Abadi, and Avi Silberschatz. "HadoopDB in Action: Building Real World Applications." In *SIGMOD*, pp. 1111–1114, 2010.

[ANB11] Tom J. Ameloot, Frank Neven, and Jan Van den Bussche. "Relational Transducers for Declarative Networking." In *PODS*, pp. 283–292, 2011.

[ant15] "antlr." `http://www.antlr.org/`, 2015.

[AOT03] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. "The Deductive Database System LDL++." *TPLP*, **3**(1):61–94, 2003.

[apa15a] "Apache Flink." `http://flink.apache.org`, 2015.

[apa15b] "Apache Spark." `http://spark.apache.org`, 2015.

[AU12] Foto N. Afrati and Jeffrey D. Ullman. "Transitive closure and recursive Datalog implemented on clusters." In *EDBT*, pp. 132–143, 2012.

[AXL15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. "Spark SQL: Relational Data Processing in Spark." In *SIGMOD*, pp. 1383–1394, 2015.

[Ban86] Francois Bancilhon. "Naive Evaluation of Recursively Defined Relations." In Michael L Brodie and John Mylopoulos, editors, *On Knowledge Base Management Systems*, pp. 165–178. Springer-Verlag, 1986.

[BBC12] Yingyi Bu, Vinayak R. Borkar, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. "Scaling Datalog for Machine Learning on Big Data." *CoRR*, **abs/1203.0160**, 2012.

[BCG11] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. "Hyracks: A flexible and extensible foundation for data-intensive computing." In *ICDE*, pp. 1151–1162, 2011.

[BEG11] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. "Jaql: A Scripting Language for Large Scale Semistructured Data Analysis." *PVLDB*, **4**(12):1272–1283, 2011.

[BEH10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. "Nephele/PACTs: a programming model and execution framework for web-scale analytical processing." In *SoCC*, pp. 119–130, 2010.

[BHB12] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. "The HaLoop approach to large-scale iterative data analysis." *VLDB Journal*, **21**(2):169–190, 2012.

[BMS86] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. "Magic Sets and Other Strange Ways to Implement Logic Programs." In *PODS*, pp. 1–15, 1986.

[cas15a] "Cascading." `http://www.cascading.org`, 2015.

[cas15b] "Cascalog." `http://cascalog.org`, 2015.

[CDD09] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. "MAD Skills: New Analysis Practices for Big Data." *PVLDB*, **2**(2):1481–1492, 2009.

[CGK90] Danette Chimenti, Ruben Gamboa, Ravi Krishnamurthy, Shamim A. Naqvi, Shalom Tsur, and Carlo Zaniolo. "The LDL System Prototype." *IEEE Trans. Knowl. Data Eng.*, **2**(1):76–90, 1990.

[CM90] Mariano P Consens and Alberto O Mendelzon. "Low complexity aggregation in GraphLog and Datalog." In *ICDT*, pp. 379–394, 1990.

[CMA12] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. "Logic and lattices for distributed programming." In *SoCC*, 2012.

[Cod72] E. F. Codd. "Relational Completeness of Data Base Sublanguages." *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[CW89] S. Cohen and O. Wolfson. "Why a Single Parallelization Strategy is Not Enough in Knowledge Bases." In *PODS*, pp. 200–216, 1989.

[DG04] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In *OSDI*, pp. 137–150, 2004.

[EF10] Jason Eisner and Nathaniel Wesley Filardo. "Dyna: Extending Datalog for Modern AI." In *Datalog Reloaded*, pp. 181–220, 2010.

[ELZ10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. "Twister: a runtime for iterative MapReduce." In *HPDC*, pp. 810–818, 2010.

[ETK12] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. "Spinning Fast Iterative Data Flows." *PVLDB*, **5**(11):1268–1279, 2012.

[fas15] "fastutil." `http://fastutil.di.unimi.it/`, 2015.

[FKR12] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. "Towards a unified architecture for in-RDBMS analytics." In *SIGMOD*, pp. 325–336, 2012.

[FPC09] Eric Friedman, Peter Pawlowski, and John Cieslewicz. "SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions." *PVLDB*, **2**(2):1402–1413, August 2009.

[FPL08] Wolfgang Faber, Gerald Pfeifer, Nicola Leone, Tina Dell'Armi, and Giuseppe Ielpa. "Design and implementation of aggregate functions in the DLV system." *TPLP*, **8**(5-6):545–580, 2008.

[FPL11] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. "Semantics and complexity of recursive aggregates in answer set programming." *Artif. Intell.*, **175**(1):278–298, 2011.

[GAK12] Todd J. Green, Molham Aref, and Grigoris Karvounarakis. "LogicBlox, Platform and Language: A Tutorial." In *Datalog 2.0*, pp. 1–8, 2012.

[Gel92] Allen Van Gelder. "The Well-Founded Semantics of Aggregation." In *PODS*, pp. 127–138, 1992.

[GGZ91] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. "Minimum and Maximum Predicates in Logic Programming." In *PODS*, pp. 154–163, 1991.

[gir15] "Apache Giraph." `http://giraph.apache.org`, 2015.

[GKS91] Sumit Ganguly, Ravi Krishnamurthy, and Abraham Silberschatz. "An analysis technique for transitive closure algorithms: A statistical approach." In *ICDE*, pp. 728–735, 1991.

[GLG12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs." In *OSDI*, pp. 17–30, 2012.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. "Maintaining Views Incrementally." In *SIGMOD*, pp. 157–166, 1993.

[GMT04] Fosca Giannotti, Giuseppe Manco, and Franco Turini. "Specifying Mining Algorithms with Iterative User-Defined Aggregates." *IEEE Trans. Knowl. Data Eng.*, **16**(10):1232–1246, 2004.

[GNC09] Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. "Building a HighLevel Dataflow System on top of MapReduce: The Pig Experience." *PVLDB*, **2**(2), 2009.

[GST90] Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. "A Framework for the Parallel Processing of Datalog Queries." In *SIGMOD*, pp. 143–152, 1990.

[GST92] Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. "Parallel Bottom-Up Processing of Datalog Queries." *J. Log. Program.*, **14**(1&2):101–126, 1992.

[gtg15] "GTgraph." http://www.cse.psu.edu/~kxm85/ software/GTgraph, 2015.

[GXD14] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. "GraphX: Graph Processing in a Distributed Dataflow Framework." In *OSDI*, pp. 599–613, 2014.

[GZ01] Sergio Greco and Carlo Zaniolo. "Greedy Algorithms in Datalog." *TPLP*, **1**(4):381–407, 2001.

[GZG92] Sergio Greco, Carlo Zaniolo, and Sumit Ganguly. "Greedy by Choice." In *PODS*, pp. 105–113, 1992.

[had15] "Apache Hadoop." http://hadoop.apache.org, 2015.

[Hel10] Joseph M. Hellerstein. "The declarative imperative: experiences and conjectures in distributed logic." *SIGMOD Record*, **39**(1):5–19, 2010.

[HRS12] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. "The MADlib Analytics Library or MAD Skills, the SQL." *PVLDB*, **5**(12):1700–1711, 2012.

[IBY07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks." In *EuroSys*, pp. 59–72, 2007.

[jun15] "junit." http://junit.org/, 2015.

[KBG12] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. "GraphChi: Large-scale Graph Computation on Just a PC." In *OSDI*, pp. 31–46, 2012.

[Kol91] Phokion G. Kolaitis. "The expressive power of stratified logic programs." *Info. and Computation*, **90**(1):50–66, 1991.

[LBG12] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud." *PVLDB*, **5**(8):716–727, 2012.

[LCG09] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. "Declarative Networking." *Commun. ACM*, **52**(11):87–95, 2009.

[LFV12] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. "The Vertica Analytic Database: C-Store 7 Years Later." *PVLDB*, **5**(12):1790–1801, 2012.

[LGK10] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. "GraphLab: A New Framework For Parallel Machine Learning." In *UAI*, pp. 340–349, 2010.

[LLM98] Georg Lausen, Bertram Ludäscher, and Wolfgang May. "On Active Deductive Databases: The Statelog Approach." In *Transactions and Change in Logic Databases*, pp. 69–106, 1998.

[log15] "log4j." `http://logging.apache.org/`, 2015.

[MAB10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-scale Graph Processing." In *SIGMOD*, pp. 135–146, 2010.

[MIG12] Svilen R. Mihaylov, Zachary G. Ives, and Sudipto Guha. "REX: Recursive, Delta-based Data-centric Computation." *PVLDB*, **5**(11):1280–1291, July 2012.

[MMI13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow System." In *SOSP*, pp. 439–455, 2013.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. "The Magic of Duplicates and Aggregates." In *VLDB*, pp. 264–277, 1990.

[MS95] Inderpal Singh Mumick and Oded Shmueli. "How Expressive is Stratified Aggregation?" *Annals of Mathematics and AI*, **15**:407–435, 1995.

[MSZ13a] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. "A declarative extension of horn clauses, and its significance for datalog and its applications." *TPLP*, **13**(4-5):609–623, 2013.

[MSZ13b] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. "Extending the Power of Datalog Recursion." *VLDB J.*, **22**(4):471–493, 2013.

[OKH13] Makoto Onizuka, Hiroyuki Kato, Soichiro Hidaka, Keisuke Nakano, and Zhenjiang Hu. "Optimization for iterative queries on MapReduce." *PVLDB*, **7**(4), 2013.

[PBM99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. "The PageRank Citation Ranking: Bringing Order to the Web." Technical Report 1999-66, Stanford InfoLab, November 1999.

[PL10] Russell Power and Jinyang Li. "Piccolo: Building Fast, Distributed Programs with Partitioned Tables." In *OSDI*, pp. 1–14, 2010.

[pro15] "Project Daytona." `http://research.microsoft.com/en-us/projects/daytona`, 2015.

[RPB13] Joshua Rosen, Neoklis Polyzotis, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. "Iterative MapReduce for Large Scale Machine Learning." *CoRR*, **abs/1303.3517**, 2013.

[RS92] Kenneth A. Ross and Yehoshua Sagiv. "Monotonic Aggregation in Deductive Databases." In *PODS*, pp. 114–126, 1992.

[sca15] "Scalding." `http://twitter.com/scalding`, 2015.

[SGL13] Jiwon Seo, Stephen Guo, and Monica S. Lam. "SociaLite: Datalog extensions for efficient social network analysis." In *ICDE*, pp. 278–289, 2013.

[SKH12] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. "Optimizing Large-Scale Semi-Naïve Datalog Evaluation in Hadoop." In *Datalog*, pp. 165–176, 2012.

[SL91] Jürgen Seib and Georg Lausen. "Parallelizing Datalog Programs by Generalized Pivoting." In *PODS*, pp. 241–251, 1991.

[SPS13] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. "Distributed Socialite: A Datalog-based Language for Large-scale Graph Analysis." *PVLDB*, **6**(14):1906–1917, 2013.

[SR91] S. Sudarshan and Raghu Ramakrishnan. "Aggregation and Relevance in Deductive Databases." In *VLDB*, pp. 501–511, 1991.

[SWL13] Bin Shao, Haixun Wang, and Yatao Li. "Trinity: A Distributed Graph Engine on a Memory Cloud." In *SIGMOD*, pp. 505–516, 2013.

[SZZ13] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. "Graph Queries in a Next-Generation Datalog System." *PVLDB*, **6**(12):1258–1261, 2013.

[TBC13] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. "From "Think Like a Vertex" to "Think Like a Graph"." *PVLDB*, **7**(3):193–204, 2013.

[TSJ09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. "Hive - A Warehousing Solution Over a Map-Reduce Framework." *PVLDB*, **2**(2):1626–1629, 2009.

[Van86] Allen Van Gelder. "A Message Passing Framework for Logical Query Evaluation." In *SIGMOD*, pp. 155–165, 1986.

[Van93] Allen Van Gelder. "Foundations of Aggregation in Deductive Databases." In *DOOD*, pp. 13–34, 1993.

[WBH15] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. "Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines." *PVLDB*, **8**(12):1542–1553, 2015.

[WCR11] Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. "Machine learning in ScalOps, a higher order cloud computing language." In *BigLearn*, December 2011.

[WO90] Ouri Wolfson and Aya Ozeri. "A New Paradigm for Parallel and Distributed Rule-Processing." In *SIGMOD*, pp. 133–142, 1990.

[WS88] Ouri Wolfson and Abraham Silberschatz. "Distributed Processing of Logic Programs." In *SIGMOD*, pp. 329–336, 1988.

[YIF08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language." In *OSDI*, pp. 1–14, 2008.

[YYT10] Christopher Yang, Christine Yen, Ceryen Tan, and Samuel Madden. "Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database." In *ICDE*, pp. 657–668, 2010.

[YZ14] Mohan Yang and Carlo Zaniolo. "Main Memory Evaluation of Recursive Queries on Multicore Machines." In *IEEE Big Data*, pp. 251–260, 2014.

[ZAO93] Carlo Zaniolo, Natraj Arni, and KayLiang Ong. "Negation and Aggregates in Recursive Rules: the LDL++ Approach." In *DOOD*, pp. 204–221, 1993.

[ZBW12] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. "SCOPE: Parallel Databases Meet MapReduce." *VLDB Journal*, **21**(5):611–636, October 2012.

[ZCD12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing." In *NSDI*, 2012.

[ZCF97] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard Thomas Snodgrass, V. S. Subrahmanian, and Roberto Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.

[ZDL13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized streams: fault-tolerant streaming computation at scale." In *SOSP*, pp. 423–438, 2013.

[ZGG11] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. "PrIter: A Distributed Framework for Prioritized Iterative Computations." In *SOCC*, pp. 13:1–13:14, 2011.

[ZGG12] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. "iMapReduce: A Distributed Computing Framework for Iterative Computation." *Journal of Grid Computing*, **10**(1):47–68, 2012.

[ZW99] Carlo Zaniolo and Haixun Wang. "Logic-Based User-Defined Aggregates for the Next Generation of Database Systems." In Krzysztof R. Apt, Victor Marek, Mirek Truszczynski, and David S. Warren, editors, *The Logic Programming Paradigm*, pp. 401–426. Springer Verlag, 1999.