UCLA
UCLA Electronic Theses and Dissertations

**Title**
Power, Performance and Scalability for Big Data Query Languages: The Machine Learning Challenge

**Permalink**
https://escholarship.org/uc/item/8m50s7jz

**Author**
Wang, Jin

**Publication Date**
2020

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

# Power, Performance and Scalability for Big Data Query Languages: The Machine Learning Challenge.

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Jin Wang

2020

© Copyright by

Jin Wang

2020

# ABSTRACT OF THE DISSERTATION

Power, Performance and Scalability for Big Data Query Languages:
The Machine Learning Challenge.

by

Jin Wang
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2020
Professor Carlo Zaniolo, Chair

In the Big Data era, there is a resurgence of interest in using *Datalog* to express data analysis applications that require recursive computations. However, the use of non-monotonic aggregates in recursion raises difficult semantic issues. Recent theoretical advances like monotonic aggregation and Pre-Mapability (PreM) provide the formal semantics for the usage of aggregates in recursive *Datalog* rules enabling the expression of a wide spectrum of advanced analytical tasks, such as graph analysis, data mining, machine learning and stream processing. In this dissertation, we explore opportunities and issues created by these advances, including the expressiveness of Datalog in advanced applications and their optimization to achieve superior performance and scalability.

Firstly, we find that *Datalog* serves as an efficient query language that simplifies the writing of machine learning applications and provides a unified environment for their development and deployment on multiple platforms. Following this route, we propose a declarative machine learning framework of tested effectiveness on top of Apache Spark. We present an in-depth theoretical analysis that shows how key ML algorithms can be expressed and effi-

ciently implemented by recursive *Datalog* programs that use aggregates in recursion, whereby achieving both formal and efficient operational semantics. We also present the compilation and optimization techniques we developed to support the complex recursive queries required by ML applications in distributed share-nothing architectures. Next we share some theoretical results to show that programs computing any aggregates on sets of facts of predictable cardinality are equivalent to stratified programs where the pre-computation of cardinality of the set is followed by a stratum where recursive rules only use monotonic constructs. Finally, we investigate how to improve the parallelism of semi-naive evaluation of recursive *Datalog* programs on shared-memory multi-core machines, and discuss the prototype system we have developed and the high performance levels it delivers.

The dissertation of Jin Wang is approved.

Ying Nian Wu

Guoqing Harry Xu

Junghoo Cho

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

2020

*To my mother and father*

TABLE OF CONTENTS

**1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

**2 Background** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5**

2.1 Datalog . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 Terminology for Recursive Queries . . . . . . . . . . . . . . . . . . . . 7

2.3 The PreM Property . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

**3 Declarative Machine Learning Framework** . . . . . . . . . . . . . . . . **11**

3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.2 Basics of Machine Learning . . . . . . . . . . . . . . . . . . . . . . . 14

3.3 Datalog for Machine Learning . . . . . . . . . . . . . . . . . . . . . . 15

3.3.1 Expressing ML Applications . . . . . . . . . . . . . . . . . . . . 15

3.3.2 Supporting Mini-batch Gradient Descent . . . . . . . . . . . . . . 19

3.3.3 Termination Condition . . . . . . . . . . . . . . . . . . . . . . . 21

3.4 Query Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

3.4.1 The BigDatalog System . . . . . . . . . . . . . . . . . . . . . . 23

3.4.2 Supporting Complex Recursions . . . . . . . . . . . . . . . . . . 24

3.4.3 Execution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

3.5 Performance Optimization . . . . . . . . . . . . . . . . . . . . . . . . 30

3.5.1 Eliminating Unnecessary Evaluation . . . . . . . . . . . . . . . . 31

3.5.2 Join Optimization with Replica . . . . . . . . . . . . . . . . . . . 32

3.5.3 Scheduling Optimization . . . . . . . . . . . . . . . . . . . . . . 34

3.6 Usability . . . . . . . . . . 35
3.6.1 Equivalent SQL Queries . . . . . . . . . . 36
3.6.2 ML Library with DataFrame APIs . . . . . . . . . . 37
3.7 Graphics User Interface . . . . . . . . . . 39
3.7.1 System Architecture . . . . . . . . . . 39
3.7.2 Interaction with the GUI . . . . . . . . . . 42
3.8 Experiments . . . . . . . . . . 43
3.8.1 Experimental Setup . . . . . . . . . . 43
3.8.2 End-to-end Performance . . . . . . . . . . 46
3.8.3 Results for Mini-batch GD . . . . . . . . . . 48
3.8.4 Scalability . . . . . . . . . . 49
3.8.5 Results on Dense Datasets . . . . . . . . . . 50
3.9 Related Work . . . . . . . . . . 51
3.9.1 Datalog for Machine Learning . . . . . . . . . . 51
3.9.2 Recursive Query Processing . . . . . . . . . . 52
3.9.3 Large-scale Machine Learning . . . . . . . . . . 53
3.9.4 Machine Learning and Big Data Systems . . . . . . . . . . 53
3.10 Conclusion of Chapter . . . . . . . . . . 54

**4 Semantics of Completed Aggregates in Recursion . . . . . . . . . . 55**
4.1 Introduction . . . . . . . . . . 55
4.2 Set Aggregation Semantics . . . . . . . . . . 57
4.2.1 Basic Definition of Continuous Count . . . . . . . . . . 58
4.2.2 Extension to Sum and Average . . . . . . . . . . 59

4.2.3 Group By Aggregates . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.3 The Pre-Countable Cardinality Property . . . . . . . . . . . . . . . . . . . . . . . 62
4.3.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.3.2 Semantics provided by PCC . . . . . . . . . . . . . . . . . . . . . . . 64
4.3.3 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.4 Formal Semantics of Machine Learning Applications . . . . . . . . . . . . . . 67
4.5 Conclusion of Chapter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

**5 Optimizing Parallel Recursive Datalog Evaluation on Multi-core Machines 69**

5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
5.2 Preliminary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
5.2.1 Parallel Evaluation of Datalog Programs . . . . . . . . . . . . . . . . 71
5.2.2 Overall Framework . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.3 Dynamic Coordination Strategy . . . . . . . . . . . . . . . . . . . . . . . . . 75
5.3.1 Parallel Execution Mechanism . . . . . . . . . . . . . . . . . . . . . . 75
5.3.2 The DWS Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
5.3.3 Theoretical Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.4.1 Experiment Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.4.2 End-to-end Query Time Comparison . . . . . . . . . . . . . . . . . . 90
5.4.3 Micro-Benchmarking Results . . . . . . . . . . . . . . . . . . . . . . . 91
5.4.4 Scalability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.5 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
5.5.1 Datalog Language and Evaluation . . . . . . . . . . . . . . . . . . . . 93

5.5.2 Datalog Systems and Applications . . . . . . . . . . . . . . . . . . . . 94
5.5.3 Parallel Query Evaluation . . . . . . . . . . . . . . . . . . . . . . . . 95
5.6 Conclusion of Chapter . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

**6 Conclusion and Future Work . . . . . . . . . . . . . . . . . . . . . . . . 96**

**References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98**

LIST OF FIGURES

3.1 Snippet Scala Code: BGD for Linear Regression . . . . . . . . . . . . . . . . . . . 20
3.2 Dependency between Tables in Query 4. . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3 Logical Plan of Query 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.4 Physical Plan of Query 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.5 Intra- vs. Inter-Iteration RDDs . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.6 Optimized Physical Plan . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.7 Example of DataFrame API: Logistic Regression . . . . . . . . . . . . . . . . . . . 38
3.8 Implementation with MLlib . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.9 The Overall Architecture of RaSQL . . . . . . . . . . . . . . . . . . . . . . . . 41
3.10 RaSQL System User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.11 Performance Comparison: Training with Batch Gradient Descent . . . . . . . . 46
3.12 Performance Comparison: Training with Mini-batch Gradient Descent . . . . . . 47
3.13 Scalability: Varying Data Size . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
3.14 Performance Comparison on Dense Dataset . . . . . . . . . . . . . . . . . . . . 50

5.1 Query Performance of SSSP on LiveJournal Dataset . . . . . . . . . . . . . . . . 70
5.2 DCDatalog: The Overall Architecture . . . . . . . . . . . . . . . . . . . . . . . 74
5.3 Execution Time under Different Coordination Strategies . . . . . . . . . . . . . 76
5.4 Effect of Different Coordination Strategies . . . . . . . . . . . . . . . . . . . . 91
5.5 Scalability: Datalog on Multicore Machines . . . . . . . . . . . . . . . . . . . . 92

LIST OF TABLES

3.1 Settings for ML Algorithms. For SVM, we append an extra 1/-1 for each instance to save the bias parameter; $\mu$ is a hyper-parameter which controls the weight of regularization term. Meanwhile, we use a *sign function* to deal with the derivative near 0 of L1 regularization in Lasso regression. . . . . . . . . . . . . . . . . . . . 17

3.2 Non-Linear Recursion Optimization . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.3 Effect of Replica . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

3.4 Effect of Scheduling Optimization . . . . . . . . . . . . . . . . . . . . . . . . . . 35

3.5 Statistics of Datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

5.1 Graph and Network Datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

5.2 Comparison with State-of-the-art Systems (seconds): OOM means out of memory; NS means the system does not support the corresponding query; TO means timeout . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

## ACKNOWLEDGMENTS

First and foremost, I would like to express my deep gratitude to my advisor Professor Carlo Zaniolo for his support and guidance in the whole process of my PhD studies in UCLA. As a respected scholar, Professor Zaniolo's diligence, enthusiasm, and immense knowledge sets a very good example for me. I deeply appreciate him for providing me the opportunity to study as a PhD student in UCLA and giving me freedom to try different research topics. His patience, caring and trust makes the past five years a wonderful time for me. I also want to extend my sincere thanks to my committee members Professor Junghoo Cho, Guoqing Harry Xu and Ying Nian Wu for their help and support throughout my studies.

In addition, I am extremely grateful to my frequent collaborators Mingda Li, Chunbin Lin and Jiacheng Wu, who offer tremendous help to my projects towards the dissertation. Besides, I would like to thank other collaborators who provide me wonderful suggestions and generous assistance, including but not limited to Ariyam Das, Jiaqi Gu, Youfu Li, Zhongyuan Wang and Guorui Xiao. I also want to thank all my friends and fellow students at UCLA, especially those in the ScAi lab, for their friendship and encouragement during my life as a PhD student in the past five years.

Last but not least, I am grateful to my parents Shisheng Wang and Guiping Zhang for their unconditional love. Their encouragement and understanding always help me walk through the ups and downs since the beginning of my undergraduate study. Without their constant support, I cannot finish the long and difficult journey as a student from bachelor, master to PhD.

## VITA

| | |
|---|---|
| 2007.8-2011.6 | B.E. Computer Science |
| | University of Science and Technology Beijing |
| | Beijing, China |
| 2012.8-2015.6 | M.E. Computer Science |
| | Tsinghua University |
| | Beijing, China |
| 2015.9-2020.7 | Ph.D. Computer Science |
| | University of California, Los Angeles |
| | Los Angeles, CA |

## PUBLICATIONS

**Jin Wang**, Chunbin Lin, Mingda Li, Carlo Zaniolo. Boosting Approximate Dictionary-based Entity Extraction with Synonyms. Information Sciences, Vol. 530, pages: 1-21, 2020.

**Jin Wang**, Guorui Xiao, Jiaqi Gu, Jiacheng Wu, Carlo Zaniolo. RASQL: A Powerful Language and its System for Big Data Applications. ACM International Conference on Management of Data (**SIGMOD**) 2020, pages: 2673-2676.

**Jin Wang**, Chunbin Lin. Fast Error-tolerant Location-aware Query Autocompletion. IEEE International Conference on Data Engineering (**ICDE**) 2020, pages: 1998-2001.

**Jin Wang**, Chunbin Lin, Carlo Zaniolo. MF-Join: Efficient Fuzzy String Similarity Join with Multi-level Filtering. IEEE International Conference on Data Engineering (**ICDE**) 2019, pages: 386-397.

**Jin Wang**, Chunbin Lin, Mingda Li, Carlo Zaniolo. An Efficient Sliding Window Approach for Approximate Entity Extraction with Synonyms. International Conference on Extending Database Technology (**EDBT**) 2019, pages: 109-120.

Jiaheng Lu, Chunbin Lin, **Jin Wang**, Chen Li. Synergy of Database Techniques and Machine Learning Models for String Similarity Search and Join. ACM International Conference on Information and Knowledge Management (**CIKM**) 2019, pages: 2975-2976. (Tutorial)

Ariyam Das, **Jin Wang**, Sahil M. Gandhi, Jae Lee, Wei Wang, Carlo Zaniolo. Learn Smart with Less: Building Better Online Decision Trees with Fewer Training Examples. International Joint Conference on Artificial Intelligence (**IJCAI**) 2019, pages: 2209-2215.

Ariyam Das, Youfu Li, **Jin Wang**, Mingda Li, Carlo Zaniolo. BigData Applications from Graph Analytics to Machine Learning by Aggregates in Recursion. International Conference on Logic Programming (**ICLP**) 2019, pages: 273-279.

**Jin Wang**, Zhongyuan Wang, Dawei Zhang, Jun Yan. Combining Knowledge with Deep Convolutional Neural Network for Short Text Classification. International Joint Conference on Artificial Intelligence (**IJCAI**) 2017, pages: 2915-2921.

Jiaqi Gu, **Jin Wang**, Carlo Zaniolo. Ranking Support for Matched Patterns over Complex Event Streams: the CEP-R System. IEEE International Conference on Data Engineering (**ICDE**) 2016, pages: 1354-1357.

# CHAPTER 1

# Introduction

In the era of Big Data, it is becoming more and more important to support analytical queries over ever-increasing volumes of data. There has been a large body of studies in developing efficient and scalable systems to support large-scale analytical queries, such as Microsoft Scope [ZBW12], Apache Hive [TSJ10], Hyracks [BCG11], Facebook Presto [STS19], Apache Spark [ZCD12] and Apache Flink [CEF17]. The success in developing such systems has laid a good foundation for scaling up analytical queries due to their natural in-memory support for iterative applications.

The growing body of research on scalable data analytics has brought a renaissance of interest in *Datalog* because of its ability to specify declarative data-intensive applications that execute efficiently over different systems and architectures. This provides new opportunities to realize the goal of combining the rigor and power of logic in expressing queries and reasoning with the performance and scalability by which relational databases manage their data. To this end, major technical challenges must be met under the following two aspects: (i) how to express the wide spectrum of analytical applications with sufficient expressive power, and (ii) how to improve scalability and performance at the system level.

In particular, it is essential to improve the ability of *Datalog* to express advanced analytical applications. To this end, a common trend is to enable the usage of aggregates satisfying particular conditions in recursions. Earlier approaches [MPR90, GGZ91, GGZ95] had primarily focused on providing a formal semantics that could accommodate the non-monotonic nature of the aggregates. However, these approaches have several constraints, including strict

prerequisites and limitations in real-life application scenarios. On the other hand, the idea of using *monotonic aggregate* proposed in [MSZ13], proved quite effective at expressing a rich set of graph and data mining algorithms [CDI18, ZYI18]. The formal semantic of such queries relies on the fact that programs satisfying the Pre-Mappability (PreM) property [ZYD17] can be transformed into equivalent aggregate-stratified programs. They enable the concise expression and efficient support of much more powerful algorithms than those expressible by programs that are stratified w.r.t. negation and aggregates. Actually, PreM of constraints provides a simple criterion that (i) the system optimizer can utilize to push constraints into recursion; and (ii) the user can write *Datalog* programs with extrema in recursion, with the guarantee that they have indeed a formal fixpoint semantics. Unfortunately, while the notion of PreM [ZYD17] works well for simple recursive queries that only use `min` and `max` aggregates, they are insufficient to deal with the classical ML applications which, along with extrema, also make extensive usage of other aggregates, such as `sum`, `count` and `average`.

As mentioned above, the second aspect of the problem and a critical factor in the success of a declarative query language, consist in the ability of its compiler/optimizer to turn queries into efficient and scalable executions. Recently many efforts have been made to develop parallel *Datalog* systems in both shared-memory and shared-nothing environments, including SociaLite [SPS13], Myria [WBH15], BigDatalog [SYI16], DeALS-MC [YSZ17] and RecStep [FZZ19] and others discussed later, that have produced promising results in many real world applications. As reported in recent studies [SYI16, FZZ19], the *Datalog* engines can outperform most special-purpose graph systems by an obvious margin. Nevertheless, existing *Datalog* engines can only handle queries with simple recursions. In real world applications such as machine learning and data mining, the queries often involve complicated recursions, i.e. mutual recursion and non-linear recursions. The heuristic solutions proposed in [SYI16] for these complex cases are limited to special cases and they can neither be applied to generalized queries nor reliably guarantee good performance. This situation calls for devising semi-naive fixpoint optimization techniques that go well beyond those tackled

in previous studies.

**Contribution** Based on the recent theoretical advances discussed above [MSZ13, ZYD17, ZYI18], we present a series of studies, ranging from theoretical analysis to practical applications, to take advantage of the recursive queries expressed with *Datalog*. The contributions of this dissertation are summarized as following:

Firstly, as machine learning (ML) becomes the core of data analysis tasks, we aim at supporting efficient ML applications with *Datalog*. We argue that declarative abstractions based on *Datalog* naturally fit for machine learning and propose a purely declarative ML framework with a Datalog query interface. We show that the use of aggregates in recursive *Datalog* programs entails a concise expression of ML applications, while providing a strictly declarative formal semantics. This is obtained by introducing simple conditions under which the semantics of recursive programs is guaranteed to be equivalent to that of aggregate-stratified ones. We further provide specialized compilation and planning techniques for semi-naive fixpoint computation in the presence of aggregates, and optimizations to accommodate for complicated recursions on distributed data platforms. To test and demonstrate these research advances, we have developed our system on top of Apache Spark. Extensive evaluations on large-scale real datasets illustrate that our approach can achieve promising performance gain while offering both increased generality and programming ease for ML applications.

Secondly, we propose the Pre-Computable Cardinality (PCC) property, to provide the formal semantics of using a complete set of aggregates, i.e. min, max, count, sum, average, in recursion. Previously the PreM property [ZYD17] had provided formal semantics for applications with `min` and `max` extremas in recursions. We find that in a recursive *Datalog* program, the evaluation of a broader scope of aggregates in recursion can be realized by monotonic constructs once the cardinality of relations on which the aggregates are evaluated can be pre-computed at a lower stratum in the recursion. As a result, while the formal semantics of stratified programs is preserved, its computation is achieved by a single-stratum

semi-naive fixpoint computation delivering great performance and scalability. This property can be used to provide formal semantics for *Datalog* programs in many advanced analytical applications, such as machine learning and data mining.

Thirdly, we developed a prototype system to boost the performance of in-memory parallel evaluation of *Datalog* programs on shared-memory multi-core machines. We devised an adaptive coordination strategy to improve the parallelism of semi-naive evaluation process of *Datalog* programs. Experimental results show that our work can significantly outperform other coordination strategies.

**Outline** The rest of the dissertation is organized as following: Chapter 2 provides some necessary background of *Datalog* and its evaluation. Chapter 3 proposes the declarative framework for machine learning with *Datalog* and SQL query interface. Chapter 4 introduces the Pre-Computable Cardinality (PCC) property that provides formal semantics of queries for machien learning and other applications. Chapter 5 described the new coordination techniques we devised for *Datalog* evaluation on multicore machine. Finally Chapter 6 concludes the whole dissertation and discusses future work.

# CHAPTER 2

# Background

## 2.1 Datalog

A Datalog program $P$ consists of a finite set of rules operating on sets of facts described by database-like schemas. A rule $r$ has the form $h \leftarrow r_1, r_2, ..., r_n$, where $h$ is the head of rule, $r_1, r_2, ..., r_n$ is the body and the comma separating atoms in the body is logical conjunction (AND). The rule head $h$ and each $r_i$ are atoms having form $p(t_1, t_2, ..., t_k)$, where $p$ is the predicate and $t_1, t_2, ..., t_k$ are terms which can be variables or constants. On occasions, We use the terms predicate, table and relation interchangeably. A rule defines a logical implication: if all predicates in the body are true, then so is the head $h$. There are two kinds of relations: (i) the base relations are defined by tables in the *EDB* (extensional database) and (ii) the derived relations are defined by the heads of rules and form the *IDB* (intentional database).

*Query 1* - Transitive Closure (TC)

$$\begin{aligned} \mathsf{r}_{1,1}: \quad tc(X,Y) &\leftarrow arc(X,Y) \\ \mathsf{r}_{1,2}: \quad tc(X,Y) &\leftarrow tc(X,Z), arc(Z,Y) \end{aligned}$$

Datalog concepts and terminology are illustrated by the Transitive Closure program in Query 1 which derives the IDB relation $tc$ from the EDB table arc representing the edges of a graph. Since the predicate $tc$ is contained in both the head and the body of rule $\mathsf{r}_{1,2}$, $tc$ is a recursive predicate and $\mathsf{r}_{1,2}$ is a *recursive rule*. The recursive predicate $tc$ is also the head predicate for $\mathsf{r}_{1,1}$ which is non-recursive and therefore provides the *base rule* in the fixpoint definition and computation of the recursive predicate. In fact the process of query

evaluation first initializes $tc$ using $\mathsf{r}_{1,1}$, and then uses $\mathsf{r}_{1,2}$ to recursively produce new $tc$ facts from the conjunction of $tc$ facts generated in previous iterations and the arc relation. Since at most one recursive goal is included in the body of any rule, Query 1 represents a case of *linear recursion*; the term *non-linear recursion* denotes instead the case where some rules contain multiple recursive goals. A Datalog program is called a *positive program* if there is no negation or aggregate in it.

The state-of-the-art method for evaluating a Datalog program is the *semi-naive* (SN) evaluation [AHV95]. SN performs the differential fixpoint computation of Datalog programs in a bottom-up manner. It starts with the application of the base rule and then iteratively applies recursive delta rules until a *fixpoint* is reached, as it will be formally defined next. The core idea of the SN optimization is that, instead of using the original rules, the evaluation can use delta rules that are based on the facts which were generated in the previous iteration step.

**Algorithm 1**: Semi-naive Evaluation of Query 1

```
1  begin
2      δtc = arc(X, Y);
3      tc = δtc;
4      do
5          δtc' = Π_{X,Y}(δ tc(X, Z) ⋈ arc(Z, Y)) − tc;
6          tc = tc ∪ δtc';
7          δtc = δtc'
8      while δtc ≠ ∅ ;
9      return tc;
10 end
```

For example, consider how the Transitive Closure program of Query 1 is evaluated by Algorithm 1. The evaluation starts with the exit rule $\mathsf{r}_{1,1}$ (line: 2) and then iterates with the recursive rule $\mathsf{r}_{1,2}$ (line: 4-8). We use tc and tc′ to denote the set of facts in the recursive

relation $tc$ at the beginning and end of the current iteration, respectively. Then the set of facts generated in the current iteration could be calculated as $\delta\mathsf{tc} = \mathsf{tc}' - \mathsf{tc}$ (line: 5). And the contents of $\mathsf{tc}$ and $\mathsf{tc}'$ are updated for the next iteration of evaluation (line: 6-7). During the evaluation of $\mathsf{r}_{1,2}$ in the next iteration, instead of using the whole relation $\mathsf{tc}(X, Z)$, it just joins $\delta\mathsf{tc}(X, Z)$ with $\mathsf{arc}(Z, Y)$. In this example, the fixpoint is reached when $\delta\mathsf{tc} = \emptyset$ (line: 8). SN has been widely applied in evaluating recursive Datalog programs. Simple SN extensions for recursive queries with aggregates have been proposed for the single-node [SYZ15], multi-core [YSZ17] and distributed [SYI16] environments.

## 2.2 Terminology for Recursive Queries

To describe the recursive queries expressed by Datalog, we introduce some necessary terminologies from [AHV95] and [ZCF97].

**Monotonicity** The monotonicity property for the rules defining a recursive predicate ensures that the fixpoint procedure previously described produces a unique result that is the least fixpoint of the mapping defined by the rules. Rules that do not use negation or aggregates are monotonic in the lattice of set-containment: these rules can be implemented using union, select, projection, Cartesian product, natural join, i.e., the monotonic constructs of relational algebra. However, rules using negation are non-monotonic and can be used in recursive queries only when the resulting program has a s Rules using aggregates are equivalent to monotonic rules is some special cases, such as those discussed later in Chapter 4 where the aggregates are applied to relations that are completely known or can be computed prior to the processing of the recursive rules.

**Stratified Program** Given a Datalog program $P$, its *dependency graph* $G_P$ can be constructed as following: Every rule is a vertex, and an edge $\langle r_i, r_j \rangle$ appears in the graph whenever the head of $r_i$ appears in the body of $r_j$. If non-monotonic constructs are applied before $r_i$, the node corresponding to it in $G_p$ is a *negated node*. With the help of its dependency

graph, the stratification of a Datalog program can be formally stated as Definition 1.

**Definition 1** *By applying topological sorting over $G_P$, its node can be partitioned into $n$ strata $S_1, ..., S_n$ with larger $i$ in a lower stratum. The program $P$ is stratified if: $\forall$ edges $\langle r_i, r_j \rangle \in G_P$, if $r_i \in S_y$ and $r_j \in S_x$ (i) $y \geq x$ if $r_i$ corresponds to a non-negated node; (ii) $y < x$ if $r_i$ corresponds to a negated one.*

**Fixpoint Semantics** The operational semantics of a Datalog program $P$, is based on the least-fixpoint semantics of Horn Clauses. More specifically, it can be computed by iterating over the Immediate Consequence Operator (ICO) defined by the rules of the program. The ICO of program $P$ is denoted as $T_P(I)$, where $I$ is any Herbrand interpretation of $P$. If $P$ is a positive program, $T_P(I)$ would be a monotonic continuous mapping in the lattice of set-containment that the interpretation $I$ belongs to. Then we have the well-known property of least fixpoint [ZCF97]: A unique minimal solution of the equation $I = T_P(I)$ always exists and it is known as the least-fixpoint of $T_P$ denoted as $lfp(T_P)$. $lfp(T_P)$ defines the *formal semantics* of $P$. Finally, the fixpoint semantics can be formally defined as Definition 2.

**Definition 2** *For an ICO $T$, the operational semantics of a program can be described with $T^{\uparrow\omega}(\emptyset)$ as following: $\omega$ is the first infinite ordinal; $T^{\uparrow 0}(\emptyset) = \emptyset$ and $T^{\uparrow n+1}(\emptyset) = T(T^{\uparrow n}(\emptyset))$. Then $T^{\uparrow\omega}(\emptyset)$ denotes the union of $T^{\uparrow n}(\emptyset)$ for every $n$. Then a recursive program whose iteration converges to the final value in a finite number of steps reaches its fixpoint at the first integer $n+1$ where $T^{\uparrow n+1}(\emptyset) = T^{\uparrow n}(\emptyset)$.*

## 2.3 The PreM Property

To address the problems raised above, the Pre-Mappability(PreM) property [ZYD17] provides formal semantics for pushing extrema aggregates, i.e. `max` and `min`, into recursion while preserving the semantics of the original stratified program. As shown in Definition 3,

its definition is based on viewing a Datalog program as function $T(R)$ where $T$ is a relational algebra expression, and $R$ is the vector of relations used in the expression.

**Definition 3 (PreM)** *Given a function $T(R_1, \ldots R_k)$ defined by relational algebra and a constraint $\gamma$, $\gamma$ is said to be Pre-Mappable to $T$ if the following property holds:*

$$\gamma(T(R_1, \ldots, R_k)) = \gamma(T(\gamma(R_1), \ldots, \gamma(R_k))).$$

For instance, if $T$ denotes the union operator, and $\gamma$ denotes the `min` or `max` constraint, we can pre-map (i.e., push) $\gamma$ to the relations taking part in the union. The PREM property that has proven so useful in parallel and distributed data processing of extrema, is also critical in resolving the non-monotonic conundrum created by their presence in recursion.

In fact, if extrema in recursive programs satisfies the PREM property, those programs would produce the same results with their equivalent versions of aggregate-stratified ones, from which they have been obtained by "pushing" the `min` and `max` aggregates into recursion. Thus the SN fixpoint of the program simply provides a more efficient realization of the aggregate-stratified semantics.

*Query 2* - All Pair Shortest Path

$$\begin{array}{lrcl} \mathsf{r}_{2,1}: & spath(X,Y,D) & \leftarrow & arc(X,Y,D). \\ \mathsf{r}_{2,2}: & spath(X,Y,min\langle D\rangle) & \leftarrow & spath(X,Z,D1), arc(Z,Y,D2), \\ & & & D = D1 + D2. \end{array}$$

For example, Query 2 expresses the All Pair Shortest Path computation which identifies the shortest paths between all pairs of nodes in the graph. In rule $\mathsf{r}_{2,1}$, arc denotes all edges in a graph while $D$ is the distance between nodes $X$ and $Y$. The rule $\mathsf{r}_{2,2}$ takes arcs originating in $Z$ and appends them to previously produced paths terminating at $Z$, where the length of the new arc is $D = D1+D2$. In this process, it is safe to pre-map the `min` aggregate to $D$ as it would filter out the tuples in spath that resulted in non-minimal values of $D$. Consequently, the performance of the query would also be much more efficient than the stratified version

that only applies the `min` aggregate after the recursive iterations. More details regarding the ability of PreM to optimize graph queries are provided in [GWM19, DLW19], where efficient techniques for testing the validity of PreM for the applications at hand were also discussed. Regarding techniques for proving PreM, the interested readers can find more details in [ZYD17, DZ19]. However, the PreM property only applies to constraints with `min` and `max` aggregates. This is not the case for `sum`, `count`, `average` and other aggregates. To resolve such issues, we need to propose new approaches to deal with them in unstratified programs.

# CHAPTER 3

# Declarative Machine Learning Framework

## 3.1 Introduction

The past decades have witnessed a booming demand for large scale data analysis in diverse application domains, such as online advertisement, news recommendation, driverless cars, and voice-controlled devices. As machine learning (ML) has achieved widespread success for many data-driven analytical tasks, demand for scaling ML algorithms to ever larger datasets became inevitable. Recently, researchers from both academia and industry have devoted great efforts to building powerful distributed data processing platforms, such as Hadoop and Apache Spark, which utilize and extend the Map-Reduce computation model. The availability of such platforms provides a great opportunity for scaling up ML applications due to their natural in-memory support of advanced big-data applications. Many scalable ML libraries based on different high-level programming languages have been provided by these platforms. A number of remarkable projects underscore the significant progress of systems and applications in this area, including MLlib [MBY16], Mahout [mah] and MADlib [HRS12] etc. Although these systems and libraries ease the burden of implementing ML applications, they still impose strict requirements on developers. Specifically, it often takes considerable efforts to develop new or customize existing ML algorithms, since developers must manage details of the distributed implementations of ML algorithms over the underlying platforms, without having full control on how and when the data is accessed.

To make better use of the computing resources and simplify the development and de-

ployment, for a declarative ML framework is needed where programming can be decoupled from the underlying algorithmic and system concerns. In other words, a framework is needed that allows users to focus on the data flow instead of low level interfaces. We believe that *Datalog* is a particularly attractive choice for expressing ML algorithms because of its natural support for reasoning and recursion simplifies ML applications. Recently, a renaissance of interest has focused on Datalog because of its succinct and declarative expression of a wide spectrum of data-intensive applications, including knowledge reasoning [BSG18], data center management [ZAC19] and social network [SPS13] etc. A common trend in the new generation of Datalog applications is the usage of aggregates in recursion, since they enable the concise expression and efficient support of much more powerful programs than those expressible by ones that are stratified w.r.t. negation and aggregates. Recent theoretical advances [MSZ13, ZYD17, ZYI18] allow us to provide formal declarative semantics to the powerful recursive queries that use aggregates in recursion. These findings have outlined the promising blueprints of a declarative ML framework using Datalog.

In this paper, we propose a declarative framework for efficiently expressing a broad range of ML applications. Unlike the previous studies that rely on user defined functions (UDF) [FKR12] and those employ a hybrid imperative and declarative framework [LGG17, JLY19, LCC17], our framework is a purely declarative programs which only uses the basic logic-based constructs of Datalog. The success of a framework critically depends on the ability of the underlying engine to turn declarative queries and programs into efficient and scalable executions. To this end, we implement our ML framework on top of BigDatalog [SYI16], which is a shared-nothing Datalog engine on top of Apache Spark, to take advantage of its power in dealing with iterative computation on massive datasets. Compared with simpler recursive applications, ML applications require recursions involving more complex structures, e.g. mutual and non-linear recursion, and multiple aggregates. This calls for optimized semi-naive fixpoint computation techniques not tackled in previous studies. To address these issues, we propose a series of compilation and planning techniques to

support these powerful Datalog programs. Moreover, we further provide a number of novel optimizations to improve the overall performance for such ML workloads. Note that our proposed techniques are platform-independent: they can also be applied to other shared-nothing Datalog platforms beyond BigDatalog.

The effectiveness of Datalog in expressing ML applications is due to the great expressive power achieved by allowing the use of aggregates satisfying particular conditions in recursions. This basic idea was first proposed in [MSZ13, ZYD17], and proved quite effective at expressing a rich set of graph and data mining algorithms [CDI18, ZYI18]. The formal semantics of such queries lies in the fact that programs satisfying the Pre-Mappability (PreM) property [ZYD17] can be transformed into equivalent aggregate-stratified programs. Unfortunately, while the notions in [ZYD17] work well for the `min` and `max` constraints used in simple recursive queries, they proved insufficient to deal with the classical ML applications which, along with extrema, also make extensive usage of other aggregates, such as `sum`, `count` and `average`. In this paper, we find that ML applications tend to apply aggregates over sets of relations whose cardinality could be pre-computed ahead of time, whereby the computation of all kinds of aggregates becomes monotonic. Following this route, we provide a formal semantics for ML applications expressed in Datalog from the aspect of fixpoint computation.

As a result of these advances, this paper makes the following contributions:

- We devise a declarative ML framework with Datalog query interface. We implement our system on top of Apache Spark and, to enhance its usability, we provide DataFrame APIs that are similar to, and actually more general than, those of Apache MLlib.

- We propose a series of compilation and planning techniques to enable the efficient expression and execution of ML applications (Section 3.4). We further develop several optimizations for the recursive plans of ML workloads, including those for the distributed evaluation, join operation and job scheduling (Section 3.5).

- We evaluate our framework on several popular benchmarks. Experimental results show that our framework outperforms, by an obvious margin, existing ML libraries on Spark, and other special-purpose ML systems as well.

The rest of the paper is organized as following: Section 3.2 reviews the basics about machine learning. Section 3.3 discusses the way to express ML applications with Datalog and its advantages. Section 3.4 presents the system implementation and proposes necessary techniques to support complicated Datalog programs for ML applications. Section 3.5 further presents several optimizations from planning to execution. Section 3.6 makes discussions about the usability issues and Section 3.7 provides a graphic user interface. Section 3.8 reports the experimental results. Section 3.9 surveys the related work. Finally Section 3.10 concludes the whole chapter.

## 3.2 Basics of Machine Learning

Generally speaking, the ML problem can be formalized as following: Given a training set $\mathcal{D}$ with $n$ instances, each instance consists of a $d$-dimensional feature vector $X_i$ ($i \in [1, n]$) with the $j^{th}$ dimension as $x_{ij}$ and a numeric target $y_i$. For the regression problems, we have $y_i \in R$; while for classification problems, we have $y_i \in \{-1, 1\}$. The process of deciding the model can be formalized as an optimization problem using the given $\mathcal{D}$. We are given a function $f(\theta; X)$ that makes prediction with a given model $\theta$ on the unseen data. The objective is to find a set of parameters $\theta^*$ that minimizes the loss function $L$ on $f$, i.e. $\theta^* = argmin_\theta L(f(\theta; X), Y)$. This can be achieved with the family of first-order-gradient optimization methods, namely gradient descent (GD).

There are different ways to compute the gradient depending on the portion of training instances that is used to update the model at each iteration, namely batch gradient descent (BGD), stochastic gradient descent (SGD) and mini-batch gradient descent (MGD). As is shown in the practice of Google's SQML project [SV17], BGD is widely adopted in modern

ML on relational engines. In this paper, we start our discussion from BGD, which computes the gradients by performing a complete pass on the training data at each iteration. BGD starts from an initial model $\theta^0$ and iterates with Equation (3.1) by the increasing number of iterations $k$ until convergence is reached.

$$\theta^{k+1} = \theta^k - (\sum_{(X,y)\in\mathcal{D}} \nabla L(f(\theta^k; X), y) + \Omega(\theta^k)) \tag{3.1}$$

where $L$ is the loss function, $\nabla$ is the function to compute gradient based on $L$ and $\Omega$ is the regularization.

## 3.3 Datalog for Machine Learning

In this section, we express ML applications with Datalog and provide the formal semantics. We first describe how to write Datalog queries for ML applications in Section 3.3.1. Then we further cover the issues of supporting generalized gradient descent and identifying stop condition in Section 3.3.2 and Section 3.3.3, respectively.

### 3.3.1 Expressing ML Applications

We will next discuss how to express ML applications with Datalog. As data sparsity is ubiquitous in ML applications, many training sets are represented in the verticalized format to save space, such as those in the famous LIBSVM benchmark [lib]. For each training instance $X = \langle Id, Y, x_1, \cdots, x_d \rangle$, the verticalization process would produce at most $d$ instances $\langle Id, Y, k, x_i \rangle$ ($k \in [1, d]$) as dimensions with value 0 will be omitted. When writing the Datalog programs, we use a verticalized view $\mathit{vtrain}(\mathtt{Id}, \mathtt{C}, \mathtt{V}, \mathtt{Y})$ to denote the training set, where `Id` denotes the id of a training instance; `Y` denotes the label; `C` and `V` denote the dimension and the value along that dimension, respectively.

With such a verticalized relation, we can now write the Datalog query to describe the training process with BGD using three recursive relations:

- *model* represents the trained model in verticalized form, where each tuple contains the following three fields: `J` is the iteration counter; `C` is a dimension in the model; and `P` is the value of parameter in that dimension.
- *gradient* represents the results of gradient computed at each iteration. `G` is the gradient value of the $\mathtt{C}^{th}$ dimension in the $\mathtt{J}^{th}$ iteration.
- *predict* represents the intermediate prediction results with the model in the current iteration for each training instance. For each tuple, `J` is the iteration counter; `YP` is the predicted $y$ value for the training instance with id `Id`.

Among these steps, the gradient computation and prediction with current model can be easily represented with aggregates in recursion. Therefore, the iterative training process can be expressed with a recursive Datalog program Query 3.

*Query 3* - Batch Gradient Descent (BGD)

$$
\begin{array}{lrcl}
\mathsf{r}_{3,1}: & \mathit{model}(0, C, 0.01) & \leftarrow & \mathit{vtrain}(_, C, _, _). \\
\mathsf{r}_{3,2}: & \mathit{model}(J1, C, NP) & \leftarrow & \mathit{model}(J, C, P), \\
 & & & \mathit{gradient}(J, C, G), \\
 & & & NP = P - lr * (G/n + \Omega(P)), \\
 & & & J1 = J + 1. \\
\mathsf{r}_{3,3}: & \mathit{gradient}(J, C, sum\langle G0\rangle) & \leftarrow & \mathit{vtrain}(Id, C, V, Y), \\
 & & & \mathit{predict}(J, Id, YP), \\
 & & & G0 = g(YP, Y, V). \\
\mathsf{r}_{3,4}: & \mathit{predict}(J, Id, sum\langle Y0\rangle) & \leftarrow & \mathit{vtrain}(Id, C, V, _), \\
 & & & \mathit{model}(J, C, P), \\
 & & & Y0 = f(V, P).
\end{array}
$$

Firstly, the model is initialized according to some predefined mechanisms in $\mathsf{r}_{3,1}$ (Here we use all 0.01 as example). Then the function $f$ is used to make prediction on all training instances according to the model obtained in the previous iteration in $\mathsf{r}_{3,4}$. Next the gradient is computed by the function $g$ (derived according to the loss function $L$) using the predicted

Table 3.1: Settings for ML Algorithms. For SVM, we append an extra 1/-1 for each instance to save the bias parameter; $\mu$ is a hyper-parameter which controls the weight of regularization term. Meanwhile, we use a *sign function* to deal with the derivative near 0 of L1 regularization in Lasso regression.

| Algorithm | Predict Function $f$ | Loss Function $L$ | Gradient $g = \bigtriangledown_P L$ | Regularizer $\Omega$ |
| --- | --- | --- | --- | --- |
| Linear Regression | $YP = V * P$ | $(YP - Y)^2$ | $2 * (YP - Y) * V$ | N/A |
| Logistic Regression | $YP = \frac{1}{1 + \mathrm{e}^{-V*P}}$ | $\begin{cases} -\log(YP), & Y = 1 \\ -\log(1 - YP), & Y = 0 \end{cases}$ | $(YP - Y) * V$ | N/A |
| SVM | $YP = V * P$ | $\max(0, 1 - Y * YP)$ | $\begin{cases} -Y * V, & \text{if } Y * YP < 1 \\ 0, & \text{otherwise} \end{cases}$ | N/A |
| L2 Regularized SVM | $YP = V * P$ | $\max(0, 1 - Y * YP)$ | $\begin{cases} -Y * V, & \text{if } Y * YP < 1 \\ 0, & \text{otherwise} \end{cases}$ | $\mu * P$ |
| Lasso Regression | $YP = V * P$ | $(YP - Y)^2$ | $2 * (YP - Y) * V$ | $\mu * \mathrm{sgn}(P)$ |
| Ridge Regression | $YP = V * P$ | $(YP - Y)^2$ | $2 * (YP - Y) * V$ | $\mu * P$ |

results in $\mathsf{r}_{3,3}$. Finally, in $\mathsf{r}_{3,2}$ the model is updated w.r.t the gradients (and optional regularization $\Omega$). Here $lr$ denotes the learning rate and $n$ is the number of training instances. And the training process moves on to the next iteration (Increase $J$ by 1).

The advantage of Query 3 lies in its generality: by varying the set of functions ($f$, $g$, $\Omega$), it can support a wide spectrum of ML algorithms [1], whereby an incomplete list of ML applications that can be expressed by Query 2 is shown in Table 3.1. Besides, the Mini-batch Gradient Descent (MGD) can also be expressed with Datalog queries with minor changes on Query 3.

The output of Query 3 is the trained model. Other necessary steps in machine learning, i.e. validation and test, can be easily implemented in a similar way. Take the evaluation on a test set as example: this can be accomplished by joining a verticalized test set *vtest* with the table *model* using a process that is similar to Query 3. Furthermore, Query 3 can be easily extended to memorize the evaluation result of each training instance in a table, which can be used to calculate other metrics such as AUC, precision, recall and accuracy. To support validation sets, a verticalized *vvalidate* table can be created to compute the loss after updating the model with $\mathsf{r}_{2,2}$ in each iteration.

We further show a concrete example of training the Linear Regression model with Batch Gradient Descent as Query 4. We will use this as the running example to demonstrate our proposed techniques in the following sections.

---

[1] In this paper, we limit our discussion to the linear models and leave the issue of deep learning models as future work.

*Query 4* - BGD for Linear Regression

$$
\begin{array}{rrcl}
\mathsf{r}_{4,1}: & model(0, C, 0.01) & \leftarrow & vtrain(_, C, _, _). \\
\mathsf{r}_{4,2}: & model(J1, C, NP) & \leftarrow & model(J, C, P), \\
& & & gradient(J, C, G), \\
& & & NP = P - lr * G/n, \\
& & & J1 = J + 1. \\
\mathsf{r}_{4,3}: & gradient(J, C, sum\langle Id, G0\rangle) & \leftarrow & vtrain(Id, C, V, Y), \\
& & & predict(J, Id, YP), \\
& & & G0 = 2 * (YP - Y) * V. \\
\mathsf{r}_{4,4}: & predict(J, Id, sum\langle C, Y0\rangle) & \leftarrow & vtrain(Id, C, V, _), \\
& & & model(J, C, P), \\
& & & Y0 = V * P.
\end{array}
$$

To demonstrate the benefits of ML applications written in *Datalog*, we will compare them with Scala programs that perform direct manipulations on RDDs. Figure 3.1 shows a fragment of a Scala program that expresses the very process of Query 4 by manipulating and directly transforming the RDDs. We can observe from this process that compared with such a Scala program, the *Datalog* program shown in Query 4 is more succinct and simpler to define since it does not require the programmer to: (i) know the details of query evaluation; (ii) specify the physical plan of dataflow and make lower-level optimizations.

### 3.3.2 Supporting Mini-batch Gradient Descent

Previously we discussed how BGD can be expressed with Datalog. Here, we further show how to support Mini-batch Gradient Descent (MGD). A major challenge is due to the fact that MGD requires the training data to be randomly shuffled before every iteration, and this can be expensive in a distributed environment. To tackle this issue, we adopt the trade-off proposed in [FKR12]: instead of making random shuffles before each iteration step, the dataset is optimally shuffled once at the beginning. Then the training data is split into

```
var data = sc.parallelize(input, numParts)
.map(d => (d.label, d.feature))
var weights = Vectors.dense(initW.toArray)
var n = weights.size
var converged = false
var i = 1
while (!converged && i <= numIterations) {
val bcWeights = data.context.broadcast(weights)
val seqOp = (grad, (label, feature)) => {
var diff = dot(feature, bcWeights.value) - label
grad += dot(diff, feature)
grad
}
val combOp = (c1, c2) => {c1 += c2}
val gradientSum = data.treeAggregate( DenseVector.zeros(n))(seqOp, combOp)
weights += dot(stepSize, gradientSum / data.size)
prevWeights = currWeights
currWeights = Some(weights)
converged = isConverged(prevWeights.get, currWeights.get, tol=1e-6)
i += 1
}
weights
```

Figure 3.1: Snippet Scala Code: BGD for Linear Regression

batches and MGD can be expressed in a way that is similar to BGD.

As described above, we need to randomly shuffle the training data before the query begins. Actually, most parts of MGD are the same as in Query 2; the only difference comes from the way in which the *predict* relation is computed and used to calculate the gradient in the current iteration. To optimize decisions, here we need the hyper-parameters of (i) batch size $bs$ and (ii) cardinality of training set $n$. The total number of batches in the training set can be calculated as $n/bs$. We can recognize the batch of training instances that will involve in each iteration in the following way: Suppose at iteration $J$, it uses the $B^{th}$ batch instead of the whole dataset for training. Then given the $Id$ of a training instance, we can identify the batch it belongs to as $Id \ \% \ n \ / \ bs$. For the $J^{th}$ iteration, only training instances belonging to the $B^{th}$ batch, where $B = J \ \% \ (n \ / \ bs)$, should be involved when calculating the table *predict*. Therefore, the computation of Mini-batch Gradient Descent can be realized by replacing $\mathsf{r}_{3,4}$ with the following rule:

$$
\begin{aligned}
\mathsf{r}_{3,4'}: \quad \mathit{pred}(J, Id, sum\langle Y0\rangle) \quad \leftarrow \quad & \mathit{vtrain}(ID, C, V, _), \\
& \mathit{model}(J, C, P), \\
& Y0 = f(V, P), \\
& Id\%(n/bs) == J\%(n/bs).
\end{aligned}
$$

### 3.3.3 Termination Condition

Finally, we discuss about the termination condition of Query 4. In recursive Datalog programs, evaluation terminates when the Datalog program reaches a *fixpoint*, producing a unique minimal model. However, this model could be infinite, in which case the fixpoint computation would never terminate, as it is fact in our examples where the temporal argument $J$ ranges over an infinite time domain. As $J$ denotes the number of iterations, increasing $J$ by 1 means training for a new iteration. In this case, the delta relation of *model* relation will always be non-empty.

To address this issue, we add conditions that terminate the iterative computation when

at least one of the following conditions is satisfied:

- The number of iteration reaches a predefined maximum number $maxJ$.
- The difference between the training losses of two adjacent iterations is smaller than a predefined value $\epsilon$.

Popular ML libraries, such as MLlib, enable users to specify hyper-parameters to control termination and limit the number of iteration in a similar manner. In our programs, we can limit the number of iterations by specifying $maxJ$ and adding the condition $J \geq maxJ$ to $\mathsf{r}_{4,2}$ in Query 4, which now becomes:

$$\begin{aligned} \mathsf{r}_{4,2'}: \quad model(J1, C, NP) \quad \leftarrow \quad & model(J, C, P), grad(J, C, G), \\ & NP = P - lr * G/n, \\ & lesser(MaxJ, J+1, J1). \end{aligned}$$

Although the IF-THEN-ELSE construct is a built-in in many Datalog systems could be used to express `lesser`, that would not be satisfactory, since the semantics of IF-THEN-ELSE is defined using negation. This would take as back to the depths of the non-monotonic conundrum from which we have just managed to emerge. Therefore we use the `lesser` predicate defined as follows in these rules:

$$\begin{aligned} lesser(MJ, I, I) \quad & \leftarrow \quad I < MJ. \\ lesser(MJ, I, MJ) \quad & \leftarrow \quad I \geq MJ. \end{aligned}$$

Similar revisions of our rules will also allow us to terminate the SN computation when the difference between training losses in two successive iterations becomes smaller than a predefined value $\epsilon$.

## 3.4 Query Evaluation

In this section, we introduce the query evaluation and optimization techniques that enabled their superior performance. In this paper, we focus on providing a detailed description

of their implementation on BigDatalog along with the extensive experiments that prove their effectiveness. However, it is clear the techniques and their promising performance can be generalized to different shared-nothing Datalog systems. We first briefly introduce the BigDatalog system which our framework is built on (Section 3.4.1). Then we introduce the new techniques to deal with complex recursions (Section 3.4.2) and query execution (Section 3.4.3).

### 3.4.1 The BigDatalog System

BigDatalog [SYI16] is a Datalog language implementation on Apache Spark. It supports relational algebra, aggregation and recursion, as well as a host of declarative optimizations. BigDatalog uses and extends Spark SQL operators, and also introduces several operators implemented in the Catalyst framework so that its planning features can be used on the recursive plans of Datalog programs.

The input processed by the BigDatalog compiler includes the DDL to specify the database schema and the query for expressing ML applications. The compiler analyzes the input query and creates a logical plan from it. To resolve recursions, the compiler recognizes recursive tables and switches from the task of building the operator tree for non-recursive queries to the specialized task required by recursive queries. After recognizing the recursive references, the compiler produces the Predicate Connection Graph (PCG) [AOT03] to identify the dependency of relations within the program.

The logical plan maps the PCG to a tree containing standard relational operators and recursion operators. Such *recursion operators* are used in the logical and physical plan to process the recursive query. The plan actually consists of the following two parts: (i) The *base plan* specifies the base case of the recursion; and (ii) The *recursive plan* defines behaviors within each iteration. In this process, the aggregates and group-by columns are automatically identified for each sub-query.

The physical plan is generated by analyzing the logical plan with the Spark SQL analyzer and applying rules defined in the optimizer. The BigDatalog operators use Spark SQL row type much in the same way in which Spark SQL uses the standard relational operators. In order to support recursion, our system introduces specialized recursion and shuffle operators into the physical plan. The proper settings for shuffle operators is identified by calling on Catalyst optimizer of Spark SQL. Finally, the query plan is executed by the Spark engine using the RDDs and transformation operators such as distinct, union and subtract.

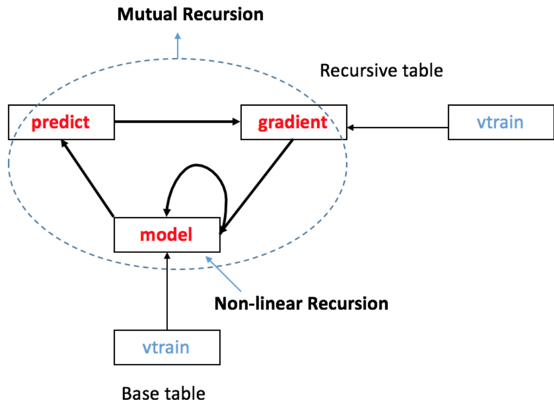



Figure 3.2: Dependency between Tables in Query 4.

### 3.4.2 Supporting Complex Recursions

#### 3.4.2.1 Challenges

Compared with simpler applications now supported by BigDatalog, ML applications require much more complex recursive queries than those discussed in [ZYI18, SYI16]. This is illustrated by the dependency graph between the four relations of Query 4 shown in Figure 3.2. We can see that the plan involves two kinds of complex recursions:

- *Mutual recursion* occurs when multiple recursive relations rely on each other to compute the result. For example, in rules $\mathsf{r}_{4,2}$ through $\mathsf{r}_{4,4}$, the recursive relations *model*, *gradient* and *predict* rely on each other and thus create a cycle which denotes a mutual

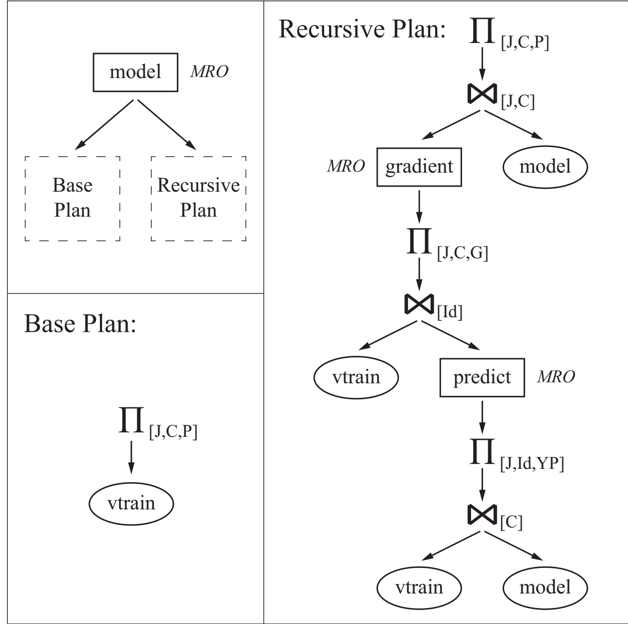



Figure 3.3: Logical Plan of Query 4

recursion in Figure 3.2.

- *Non-linear recursion* means that there are more than one recursive relation in the body of a rule. For example, rule $\mathsf{r}_{4,2}$ involves two recursive relations *model* and *gradient*.

By analyzing the PCG, the compiler recognize These two kinds of recursion and marks the rules with special tags. These tags identify the particular recursion types and the different techniques used to process them, which are described next.

#### 3.4.2.2 New Recursion Operator

To support mutual recursion, we define a special recursion operator named *Mutual Recursion Operator* (MRO), which provides a major extension to the basic recursion operator of BigDatalog that cannot be used for mutual recursion since it only allows one recursive relation in the recursive plan. MRO instead allows mutual references among multiple recursive relations by including them in the recursive plan in a cascading manner. For each set of mutually recursive relations, only one MRO has the base plan, since the base case for other

MROs is provided by the operator that precedes them in the plan.

**Example 1** *The logical plan for Query 3 is shown in Figure 3.3. The root of the plan is an MRO with both base and recursive plan. The left child is the base plan with only the* vtrain *relation representing rule* $r_{4,1}$*, which provides the base case of the mutual recursion. The right child is the recursive plan representing rules* $r_{4,2}$ *through* $r_{4,4}$*. Each MRO represents a rule within the mutual recursion. We can see that all MROs belonging to the recursive plan have a NULL base plan (omitted in Figure 3.3).*

*The corresponding physical plan is shown in Figure 3.4. It consists of operators translated from the logical plan along with the shuffle operators and their partitioning information. For example, in the recursive plan, when the join between recursive relations model and gradient is computed, both operands must be shuffled according to their join keys* $J$ *and* $C$*. The recursive plan in Figure 3.4 also shows that this join operation is followed by two more joins, each of which requires two shuffle operations. Therefore, a total of six shuffle operations is performed at each iteration.*

#### 3.4.2.3 Distributed Semi-naive Evaluation

To evaluate the program in a distributed environment, the physical plan assigns each MRO to the master node where it executes and becomes responsible for driving the distributed query evaluation. The most important step is the scheduling of shuffle operators that are injected between successive steps of the physical plan presiding to the distributed evaluation. The shuffle operators are used to re-partition the dataset in all cases where the output produced by an operator is different from that of the operator using it as input according to the execution plan. Then the BigDatalog engine utilizes fixpoint computation to drive the iterative evaluation process using the distributed version of semi-naive (DSN) evaluation.

The execution of DSN in the MapReduce framework requires the recursive relations and base relations within one stage to be co-partitioned on a given key $K$. After that,

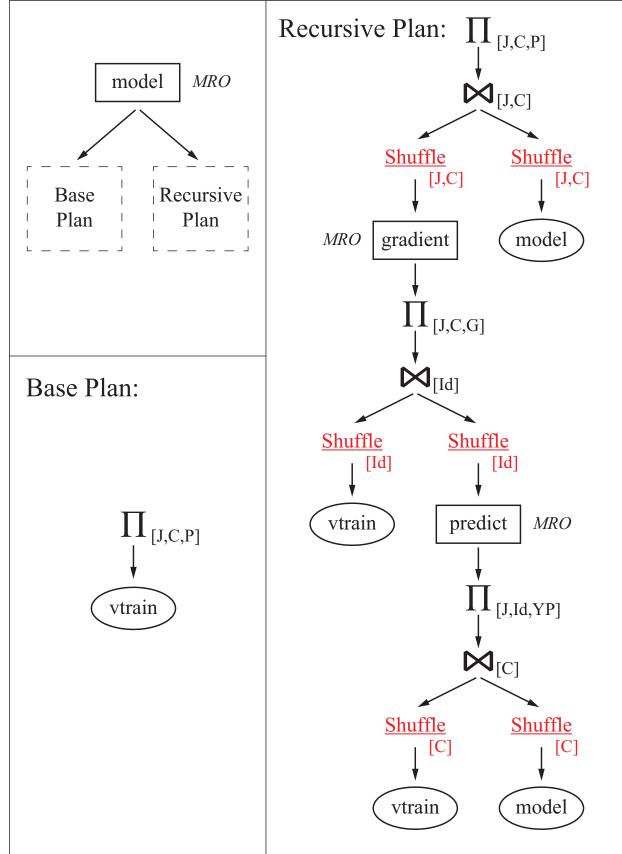



Figure 3.4: Physical Plan of Query 4

the execution goes through Map and Reduce stages. Results of the current iteration are generated in the Map stage, while the new delta and the relations needed in the next iteration are generated in the Reduce stage. Algorithm 2 describes the process in more details.

However, since programs for ML applications include non-linear and mutual recursion, we must revised the evaluation approach described above. For mutual recursion, the solution is relatively easy: One recursive relation is regarded as the driver for DSN, e.g. the *model* relation in Figure 3.4, while the others are evaluated by the MROs in the recursive plan. These extensions do not impact the techniques currently used for linear recursion.

A more complex solution is required for non-linear recursion. In fact, let $X$ and $Y$ denote two recursive relations that are involved in a non-linear recursion since they appear as goals

**Algorithm 2**: DSN Evaluation ($B$, $K$)

**Input**: $B$: The Base Relation, $K$: The partition key

**Output**: $R$: All results in the recursive table

```
1  begin
2      // δR, δR': Recursive relation (Delta)
3      Map Stage(δR, B)
4      foreach partition pair of (δR, B) do
5          emit Π(δR ⋈_{δR.K=B.K} B)
6      Reduce Stage(δR', R)
7      foreach partition pair of (δR', R) do
8          D ← δR' − R
9          R ← δR' ∪ R
10         emit D
11     δR ← Results of Base Case, R ← ∅
12     repeat
13         i ← i + 1
14         MapOutput ← MapStage(δR, B)
15         δR' ← ShuffleExchange(MapOutput, key = K)
16         δR ← ReduceStage(δR', R)
17     until δR == ∅ ;
18     return R;
19 end
```

in the body of the same rule. Then, the SN evaluation should be performed by enumerating the combination of delta relations as shown in Equation (3.2):

$$\begin{aligned}\delta(X \bowtie Y \bowtie B) = (\delta X \bowtie Y \bowtie B) \cup \\ (X \bowtie \delta Y \bowtie B) \cup (\delta X \bowtie \delta Y \bowtie B)\end{aligned} \tag{3.2}$$

where $B$ is a base relation that is optional in this process.

Therefore, unlike the case of linear recursion, we need to keep the whole recursive relations rather than just deltas in order to support non-linear recursion in DSN. During the evaluation, the steps described in line 7 to 11 of Algorithm 2 should be replaced with the operations defined Equation (3.2) in order to support non-linear recursion. Similar observations also apply when computing aggregates in recursion.

**Example 2** *For the example at hand, we can see that non-linear recursion appears in rule* $r_{4,2}$ *of Query 4 where the model relation in the head is obtained by joining model and gradient on the keys* $J$ *and* $C$*. Then the delta relation of* $r_{4,2}$ *should be computed as the union of* $model \bowtie \delta gradient$*,* $\delta model \bowtie gradient$ *and* $\delta model \bowtie \delta gradient$*. Therefore, as shown in Figure 3.4, it keeps the whole relation instead of only the delta in our physical plans.*

### 3.4.3 Execution

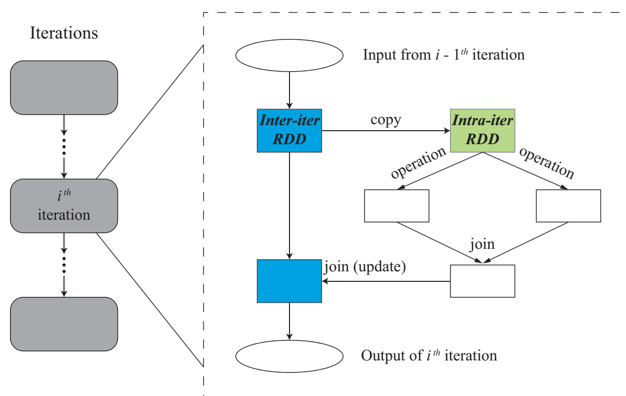



Figure 3.5: Intra- vs. Inter-Iteration RDDs

To avoid data redundancy in the process of SN evaluation, BigDatalog [SYI16] adopted the SetRDD mechanism for executing recursive queries in Spark. SetRDD stores distinct rows of data into a HashSet data structure to optimize the execution of set operators in the DSN. Thus, SetRDD is made mutable under the union operation, which saves system memory by not copying redundant data from up-stream RDDs. However, this optimization may not work when dealing with non-linear recursion: According to the mechanism of SetRDD,

when a recursive relation is referenced in one rule, its corresponding RDD would be modified by the *set union* and *set difference* operations. However, in case of non-linear recursion, a recursive relation can be referenced more than once within each iteration. Then if the recursive relation has been modified by one rule, when it is evaluated by another rule in the same iteration, its RDD is no longer the same as it was before the first evaluation, whereby the execution results would be incorrect.

To address this issue, we propose a smart strategy to divide the RDDs into *Intra-Iteration* and *Inter-Iteration* ones. Thus, for non-linear recursion, we are able to identify when the RDDs will be re-used in the same iteration. If so, we classify it as *Intra-Iteration* RDD and treat it as immutable, i.e. we generate a new RDD by copying data from the up-stream one. But when an RDD will only be used in the next iteration, we classify it as an *Inter-Iteration* RDD and process it as SetRDD to save memory.

**Example 3** *Figure 3.5 shows the series of RDDs generated in the execution step of Query 4. Here the green rectangles denote Intra-Iteration RDDs while the blue dashed ones denote Inter-Iteration ones. We are aware that in the $i^{th}$ iteration, model is updated by rule $r_{4,2}$, which would be used in the $i+1^{th}$ iteration. Meanwhile, this table is also used in rule $r_{4,4}$ that updates predict. Therefore, the RDD of model generated by $r_{4,2}$ should be* Inter-Iteration *while that used in $r_{4,4}$ should be* Intra-Iteration.

## 3.5 Performance Optimization

In this section, we present several techniques that have proven to be quite effective in optimizing the performance of our framework. To measure the effectiveness of each technique, we use the Datalog programs to train Linear Regression (Linear), Logistic Regression (Logistic) and SVM with BGD on a synthetic dataset. The data generator used here is the one proposed in a previous experimental study for ML applications [TK18]. We use the option of sparse data with density $1.67\times 10^{-6}$. The total size of training set is 40 GB. The training

process of BGD is conducted over 100 iterations.

### 3.5.1 Eliminating Unnecessary Evaluation

Table 3.2: Non-Linear Recursion Optimization

| Time (s) | Linear | Logistic | SVM |
|---|---|---|---|
| w/ elimination | 7196.4 | 7582.9 | 6814.6 |
| w/o elimination | 10358.1 | 11319.5 | 10166.7 |

For programs with non-linear recursions, we need to enumerate the combinations of delta relations as shown in Equation (3.2) when performing semi-naive evaluations. As a result, the DSN could be significantly more expensive than that with only linear recursions. An example can be observed in Query 4 where the non-linear recursion is used in $r_{4,2}$ when updating the model with the gradient computed in current iteration. The evaluation would require using the whole recursive relations *model* and *gradient* in the physical plan as shown in Figure 3.4.

As our investigation progressed from formal semantics to operational semantics, we find that while the textbook techniques for SN optimization of non-linear queries remain valid, they can be further optimized for specific ML queries. Take again Query 4 as our example: When adopting Equation (3.2) to evaluate the query, we need to count for the items $model \bowtie \delta gradient$, $\delta model \bowtie gradient$ and $\delta model \bowtie \delta gradient$ and thus need to include the full relations *model* and *gradient*. However, note that the join key between *model* and *gradient* is $\langle J, Col \rangle$. In the $i^{th}$ iteration, since tuples in *model* are from the $J-1^{th}$ iteration while those in $\delta gradient$ are from the $J^{th}$ iteration, $model \bowtie \delta gradient = \emptyset$ holds due to mismatched values of $J$. Similarly, $\delta model \bowtie gradient = \emptyset$ also holds. Therefore, we only need to evaluate the item $\delta model \bowtie \delta gradient$. As a result, the items *model* and *gradient* can be replaced with $\delta model$ and $\delta gradient$ in the physical plan, which significantly reduces the computational overhead and the network transmission caused by shuffle operations. Since

this optimization is based on the execution process of gradient descent, it can be applied for training all linear models with BGD and MGD. Figure 3.6 shows the physical plan after applying optimizations: the full relations *model* and *gradient* are replaced with delta ones.

The effect of eliminating unnecessary evaluations are shown in Table 3.2. The results show that this optimization for the SN evaluation of non-linear recursive programs for ML is quite substantial, and this is hardly a surprise given that the full relations are replaced by the delta ones at every iteration of the SN computation.

### 3.5.2 Join Optimization with Replica

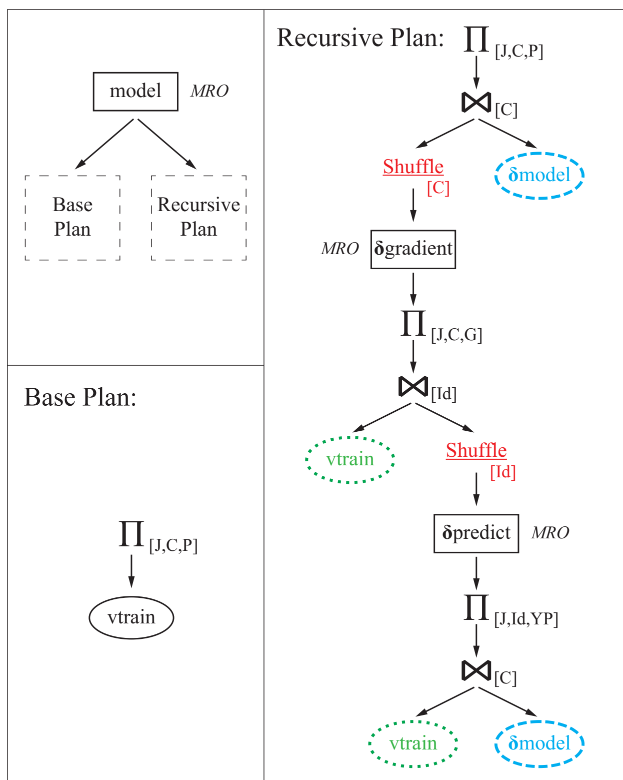



Figure 3.6: Optimized Physical Plan

For programs with linear recursion, it is often better to use broadcast join between the delta recursive relation and the base relation in the physical plan by loading the base

relation into a lookup table and shared by all workers via broadcasting. Since the overhead of broadcasting can be amortized over the recursion, this approach is rather effective for graph queries where the base table is usually much smaller than the intermediate results [SYI16]. However, the characteristics of ML workloads are totally different from those of graph queries: the size of intermediate results that participates in the computation and requires to be kept in memory is independent from the number of iterations and is relatively small: the size of *predict* is $2n$ where $n$ is the size of training data; the size of *gradient* and *model* is both $2d$, where $d$ is the dimension of a training instance [2]. In contrast, the base relation, i.e. the training set, tends to be very large. What's more, the size of base relation always exceeds the maximum memory of a single worker, which would make the broadcast join not applicable. As a result, the broadcast joins that proved so effective on graph queries will encounter serious problem on ML workloads. Consequently, there would be multiple shuffle operations per iteration on the base relation, which brought significant overhead for the overall performance. Recall that shuffle operations on the base relation happens when the base relation is joined with recursive ones on different keys. For example, in $\mathsf{r}_{4,3}$ *vtrain* needs to be joined with *predict* on the key `Id`; and in $\mathsf{r}_{4,4}$ the join key between *vtrain* and *model* becomes `C`. Here the *vtrain* relation would be shuffled twice.

To address this issue, our framework instead adopts a smart-replica optimization approach that makes careful trade-offs between memory usage and join performance. We find that the shuffle operations can be avoided by *making replicas of the base relation partitioned by different keys on the same worker*. Specifically, in above example we just make two different replicas of the *vtrain* relation on all workers: one is partitioned by the key *Id* and the other is partitioned by $C$ [3]. Then the former will be used in $\mathsf{r}_{4,3}$ while the latter will be used in $\mathsf{r}_{4,4}$. The green dotted items in Figure 3.6 are relations where the shuffle operations

[2] The total size of intermediate results would be $nJ$ for *predict* and $dJ$ for *gradient* and *model*. Results from older iterations would be dumped into disk for the sake of crash recovery.

[3] The distribution of replicas partitioned by different keys might be different on the same worker

can be avoided by making replicas of *vtrain*. As we can see, two shuffle operations could be saved compared with the original physical plan in Figure 3.4.

We also want to point out that the space overhead brought by replicas is tolerable. The essence of broadcast join is to trade the memory for join performance. Since the whole base relation is transmitted, the memory overhead on each worker would be the size of the base relation. Meanwhile, the memory overhead of our replica mechanism is the size of base table divided by the number of workers on average. It has similar benefit in accelerating join processing as broadcast join does while avoiding its shortcoming of memory consumption. Furthermore, the decision of making replicas can be made automatically: The fact that the base relation needs to participate in join operations on different keys can be recognized in the process of formalizing the logical plan. Thus the usage of replicas will be decided before actual physical plan is generated. Note that the Spark APIs cannot make such optimizations since the program is directly expressed in terms of physical operations.

Table 3.3: Effect of Replica

| Time (s) | Linear | Logistic | SVM |
|---|---|---|---|
| w/ replica | 7196.4 | 7582.9 | 6814.6 |
| w/o replica | 22664.9 | 26312.3 | 20660.0 |

The effects of applying the replica mechanism are shown in Table 3.3. We can see that with the help of replica mechanism, it achieves a performance gain of 3X to 3.4X. This underscores the considerable amount of shuffle operations that are removed from all iterations because of our carefully designed replica mechanism.

### 3.5.3 Scheduling Optimization

As illustrated in [WS88], recursive queries that can be compiled into decomposable plans could potentially benefit from a well-chosen partition strategy. In such cases, the produced RDDs preserve the original partition of input recursive table. Then the executor on the

same partition can continue to work without global synchronization. Consequently, the shuffle operations could be saved. The correctness of this property can be guaranteed by the replica mechanism on base relations even if the join key will change for the next operator. The blue dashed items in Figure 3.6 are the shuffle operations that can be saved by the scheduling optimizations. For rule $r_{4,2}$, the shuffle operation can be removed since delta of the recursive relation *model* can be acquired locally for each worker. Similarly, in rule $r_{4,4}$, the recursive relation *model* comes from $r_{4,2}$, which has already been partitioned by the same key C. Therefore, the shuffle operation on *model* can also be removed.

Table 3.4: Effect of Scheduling Optimization

| Time (s) | Linear | Logistic | SVM |
|---|---|---|---|
| w/ optimization | 7196.4 | 7582.9 | 6814.6 |
| w/o optimization | 7961.0 | 8339.2 | 7719.7 |

Table 3.4 shows the effect of scheduling optimizations. The overall performance is improved over the un-optimized approach by approximately 1.2X. Actually the elimination of shuffle operations in $r_{4,4}$ can be done automatically once the replica mechanism is applied. Therefore, the performance gain brought by scheduling optimization is not so obvious compared with the other two optimizations described above.

## 3.6 Usability

n this section, we discuss the usability issues of our proposed framework. We first introduce how to express the ML applications with SQL queries that are equivalent to the *Datalog* ones in Section 3.6.1. Next we propose a library of the *Datalog* queries for ML with DataFrame APIs in Section 3.6.2.

### 3.6.1 Equivalent SQL Queries

SQL has delivered great benefits in relational DBMS and big data platforms due to its declarative nature and portability. We show here that SQL can support many ML applications by providing SQL queries that have equivalent semantics to the Datalog ones introduced above. This represent an important extension to the RaSQL language and its system [GWM19] which supported aggregates in recursion by introducing a simple extension in the syntax of the SQL:2003 SQL standards. Specifically, RaSQL supports basic aggregates, i.e. min, max, sum, count, in recursion by minimal extensions of the Common Table Expressions (CTE) used by current SQL standard. The basic syntax of RaSQL is shown below.

```
WITH [recursive] VIEW1 (v1_column1, v1_column2, ...)
AS (SQL-expression11) UNION (SQL-expression12) ...,
[recursive] VIEW2 (v2_column1, v2_column2, ...)
AS (SQL-expression21) UNION (SQL-expression22) ...
SELECT ... FROM VIEW1 | VIEW2 | ...
```

WITH RECURSIVE construct of RaSQL

The CTE starts with the keyword “WITH RECURSIVE”, which is followed by definitions of the recursive view. The view content is defined by a union of sub-queries, which define the *base table* and *recursive table*. This is similar to the base and recursive relations of Datalog. Here a table is the base table if its FROM clause definition does not refer to any recursive CTE; otherwise it is a recursive table. The RaSQL query that is equivalent with Query 4 is shown in Query 5.

Such RaSQL queries for ML applications can be compiled into Spark SQL operators and recursive operators in a similar way to that discussed in Section 3.4. Moreover, such RaSQL queries can be encapsulated into a library called by DataFrame operations as MLlib did.

*Query 5* - RaSQL: BGD for Linear Regression

```
Base tables: vtrain(Id: int, C: int, V: double, Y: double)
WITH recursive model (J, C, P) AS
(SELECT 0, vtrain.C, 0.01 FROM vtrain)
UNION
((SELECT 1+m.J, m.C, m.P+2.0/n*LR*g.G
FROM model AS m, gradient AS g
WHERE m.C = g.C and m.J = g.J),
recursive gradient(J, C, sum() AS G) AS
(SELECT p.J, t.C, (t.Y - p.YP)*t.V
FROM vtrain AS t, predict AS p
WHERE p.Id = t.Id),
recursive predict(J, Id, sum() AS YP) AS
(SELECT m.J, t.Id, m.P*t.V
FROM vtrain AS t, model AS m
WHERE t.C = m.C)),
SELECT * FROM model
```

### 3.6.2 ML Library with DataFrame APIs

To improve usability and attract a wide participation by data scientists, we further encapsulate the *Datalog* queries for ML algorithms in a more elegant and succinct library using DataFrame APIs. Currently such a library can support all queries introduced in Section 3.3.1. With the help of such a library, users can express the whole process of machine learning using the *Datalog* queries introduced above where the hyper-parameters and data source can be specified in a similar way as MLlib does. Next we illustrate the basic usage of our API with a running example in Figure 3.7.

The example in Figure 3.7 expresses the process of training a Logistic Regression classifier

```
val session = DatalogMLlibSession.builder()
.appName("LR") .master("local[*]")
.getOrCreate()
// Import data.
var Vschema =
StructType(List(StructField("Id", IntegerType, true),
StructField("C", IntegerType, true),
StructField("V", DoubleType, true),
StructField("Y", IntegerType, true)))
var df = spark.read.format("csv")
.option("header", "false").schema(Vschema)
.load("dataDTrain")
// Training on the input relation df.
import edu.ucla.cs.wis.bigdatalog.spark.DatalogMLlib.
{DL_LogisticRegression, DL_LogisticRregressionTransformer}
val lr = new DL_LogisticRegression().setMaxIter(10)
val lrModel = lr.fit(df, session)
// Testing with pre-trained model.
var test = spark.read.format("csv")
.option("header", "false").schema(Vschema)
.load("dataDTest")
val lrPredict = new DL_LogisticRregressionTransformer()
val prediction = lrPredict.transform(lrModel, test, session)
```

Figure 3.7: Example of DataFrame API: Logistic Regression

on the training data dataDTrain, and making prediction on the test data, dataDTest. The two datasets are stored in a verticalized view with Vschema (Id, C, V, Y) as introduced in Section 3.3.1. To make use of the *Datalog* programs for machine learning, we first construct a working environment, i.e. DatalogMLlibSession for our library of machine learning algorithm (line: 1 to 3). Then, we load the training data to a Dataframe df. After importing the required training and predicting functions for Logistic Regression (line: 14 to 15), we could

build executable objects for training lr (line 16) and predicting lrPredict (line: 22). The lr object wraps all the logical rules and required relations (e.g. parameters with default value 0) of the *Datalog* implementation for Logistic Regression. When initializing lr, users can exploit the built-in functions to set the hyper-parameters, including maximum number of iterations, the method used for parameter initialization, and many others. After fitting the model to df, the lrPredict object could make predictions on the testing instances with the pre-trained model, lrModel. In both the fitting and predicting processes, the information of *Datalog* execution runtime can be obtained by using session as an input argument, which is same as the practice of MLlib.

For the sake of comparison, we also show how Apache Spark MLlib will be used to implement the above example. The snippet code is shown in Figure 3.8. The pipeline of functionalities is very similar to that of our APIs; this will make it much easier using the DataFrame APIs in our library for those who are already familiar with MLlib. Although there are minor differences in the aspects of data formatting and usage of some public functions, e.g. transform and assembler, the expression of MLlib and our library are very similar and both user-friendly.

## 3.7 Graphics User Interface

In this section, we propose a graphics user interface for the RaSQL system, which can also ease the expression of ML applications. We first introduce the system architecture in Section 3.7.1. Then we show how to make interaction with the system via the graphics user interface in Section 3.7.2.

### 3.7.1 System Architecture

The overall architecture of RaSQL system is shown in Figure 3.9. It is built on top of Apache Spark (denoted by red dashed rectangle) and consists of three components:

```
val session = SparkSession.builder().appName("LR")
.master("local[*]").getOrCreate()
// Import data.
var schema = StructType(List(StructField("X1", IntegerType, true), StructField("X2", IntegerType,
    true),
StructField("X3", DoubleType, true),
StructField("label", IntegerType, true)))
var df = spark.read.format("csv").option("header", "false").schema(schema).load("dataSTrain")
// Training on the input relation df.
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.VectorAssembler
val assembler = new VectorAssembler()
.setInputCols(Array("X1", "X2", "X3"))
.setOutputCol("features")
val lr = new LogisticRegression().setMaxIter(10)
val pipeline = new Pipeline().setStages(Array(assembler, lr))
val lrModel = pipeline.fit(df)
// Testing with pre-trained model.
var test = spark.read.format("csv").option("header", "false").schema(schema).load("dataSTest")
val prediction = lrModel.transform(lrModel, test)
```

Figure 3.8: Implementation with MLlib

**Web-based User Interface** We provide a carefully designed user interface that allows users to interact with the RaSQL system. Users can type-in the RaSQL queries as well as experience other advanced features without much effort. After the query is submitted to the RaSQL engine and gets executed, the query results would be returned and displayed on the user interface. More details will be described later in Section 3.7.2.

**Query Compilation and Planning** The RaSQL engine is implemented on top of Spark SQL, which supports the latest ANSI-SQL standard and provides a comprehensive set of functions that perform the parsing, analyzing and planning of SQL queries. However, cur-

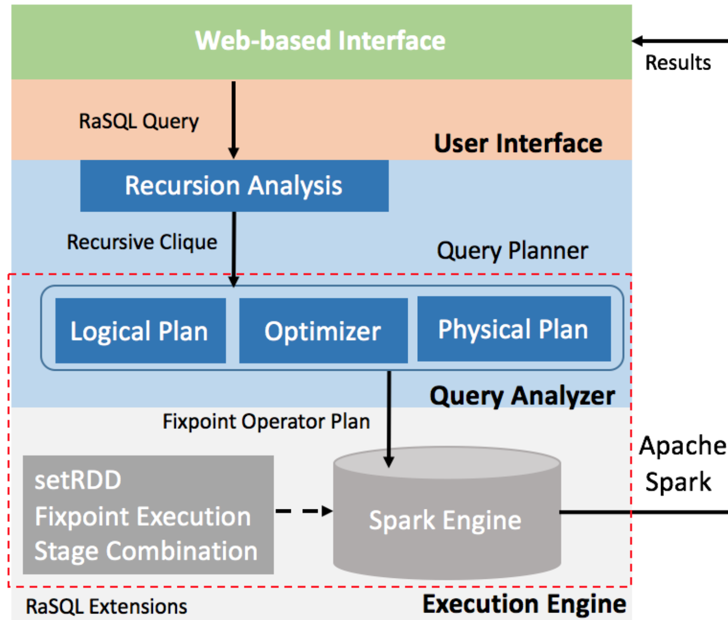



Figure 3.9: The Overall Architecture of RaSQL

rently the Spark SQL compiler cannot support recursive queries. To address this issue, we propose novel compilation and query planning techniques, The RaSQL queries will first be compiled into a *Recursive Clique* driven by a new *Recursive Operator* which consists of a base plan and a recursive plan by leveraging the techniques from *Datalog* researches [AOT03]. Then it will further be translated into recursive logical and physical plans, which consists both relational operator supported by Spark SQL and recursive operators proposed by us. We introduce a new *fixpoint operator* in order to evaluate the recursive plan and identify the termination condition for the recursion.

**Execution Engine** The query plan is then submitted to the execution engine, which is executed by the Apache Spark engine using RDDs and transformation operators. To efficiently support the execution of recursive RaSQL queries, we modify the core of Spark to support the fixpoint operator plan and propose a new data structure *setRDD*. We also propose several techniques to improve the performance, such as stage combination and partition-aware

Figure 3.10: RaSQL System User Interface

scheduling, which are explained in [GWM19].

### 3.7.2 Interaction with the GUI

To interact with the system, it uses RaSQL web interface and the screen shots are shown in Figure 3.10. This user interface consists of four main components: (i) Query input box (top left); (ii) Result display panel (bottom left); (iii) Configuration dashboard (bottom right); (iv) Query Plan display (top right). For novice users, we provide some predefined examples which can be loaded by simply clicking the name of queries in the bottom right drop-down list. Advanced users are welcome to come up with queries by typing into the input box.

The result display panel demonstrates the query results. The top right panel visualizes the recursive clique plan of the query, which can be generated by clicking the "Analyze" button after input the query. It demonstrates how the recursive RaSQL query is analyzed.

Details of operators can be found by clicking the corresponding icons. It would help users to better understand the compilation and planning techniques of RaSQL. The configuration dashboard provides several options for user to setup in order to execute the queries. For example, users need to specify the data source from either local file or HDFS on the cloud and set the arguments (if any). Next, users can directly interact with the system and test its capabilities: Users can write the RaSQL freely by themselves on different kinds of datasets, including both graph data and training data for ML models. Users can also observe the generated recursive clique and query plans to get an experience of how the recursive queries can be handled by the RaSQL engine.

## 3.8 Experiments

### 3.8.1 Experimental Setup

Table 3.5: Statistics of Datasets

| Name | Cardinality | # Features | Size (GB) |
|---|---|---|---|
| URL | 2,396,130 | 3,231,961 | 2.1 |
| KDD10 | 19,264,097 | 29,890,095 | 4.8 |
| KDD12 | 149,639,105 | 54,686,452 | 21.1 |
| Webspam | 350,000 | 16,609,143 | 23.3 |

#### 3.8.1.1 Workloads and Datasets

We evaluate the performance of our framework on the task of training linear models via gradient descent optimizers. As is stated before, we mainly focus on BGD. But we also report the results of MGD using the method described in Section 3.3.2. Specifically, we use Linear Regression, Logistic Regressionand SVM as benchmark models in this paper.

The datasets used in the experiments are summarized in Table 3.5. Here cardinality means the number of training instances while "# Features" means the number of dimension each training instance has. We conduct experiments on 4 public datasets provided by LIBSVM [lib], a popular benchmark for evaluating linear models: URL [MSS09] is a dataset for identifying malicious URLs. KDD10 comes from Carnegie Learning and DataShop that is used in KDD Cup 2010. KDD12 [JZC16] is a CTR prediction task from KDD Cup 2012. WEBSPAM [WCP06] is a dataset of email spams. Currently we are focusing on training linear models to learn from sparse datasets, which occur frequently in real-life applications. All above selected datasets are from real world scenarios. We also show some results on dense dataset later in Section 3.8.5. Note that the cardinality of such datasets are sufficient to evaluate the systems considering the memory of all available nodes. The memory needs to hold not only the dataset but also the intermediate results and system runtime, which also the case for all the baseline systems mentioned in the following.

#### 3.8.1.2 Baselines and Metrics

As BigDatalog is implemented on top of Apache Spark, we mainly compare it against two Spark based competitors: MLlib 2.3.0 and SystemML 1.2.0, where MLlib [MBY16] is the official Spark package for machine learning [4]. As MLlib comes with an implementation with MGD, we implement BGD by setting the batch size as the cardinality of training set. SystemML [BDE16] is a state-of-the-art ML system on top of Spark using a declarative R-like language [5]. We implement the training process with BGD and MGD using its script language following the official documentation. We are also aware that there are several special-purposed machine learning systems, including TensorFlow, PyTorch, MXNet and Petuum. Due to the space limitation, we just select PyTorch [6] as the representative for

[4] https://spark.apache.org/mllib/

[5] https://systemml.apache.org/

[6] https://pytorch.org/

comparison. Other studies published on Datalog for machine learning [MVP18] and [LCC17] do not provide a good basis for comparison. This is because simple query interfaces rather than end-to-end systems are provided in [MVP18] and [LCC17], and no publicly available implementation is available for [BBC12b].

Note that the main purpose of this work is not to claim that the implementation of our proposed framework is fundamentally more efficient than other special purposed ML systems, or to argue that Datalog is more suitable than the math-like syntax interfaces have provided in other ML platforms. Instead, we aim at demonstrating that it is possible to optimize a general recursive query engine to achieve the competitive or even better performance than special-purpose ML systems in a family of ML applications.

We use execution time as the evaluation metric in the experiments. Since BGD uses all training instances in one iteration, the results regarding accuracy/loss are the same for all systems. Therefore, we only report the end-to-end query execution time for models trained with BGD. For MGD we report the results of training loss vs. training time as it was done in many previous studies of ML systems. To ensure fairness, we allocate the same number of workers/servers and sufficient memory to guarantee the performance for different platforms. We ensure that algorithms on different platforms are equivalent in terms of workload and convergence by configuring the implementation on all systems with exact the same parameters.

In the experiments, the original LIBSVM data format can be supported by our approach and also by MLlib and PyTorch. For SystemML, we converted our data format into their supported binary format following the instructions in SystemML's official documentation, and we did not include this preprocessing time into the total query time.

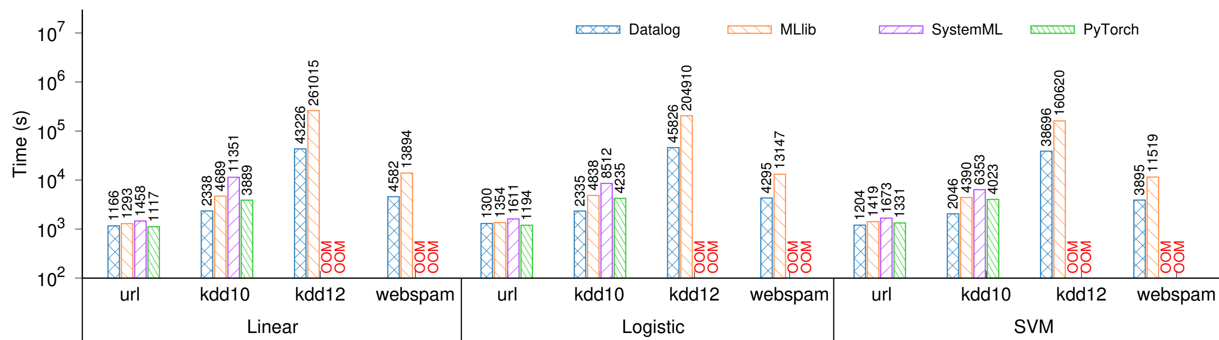


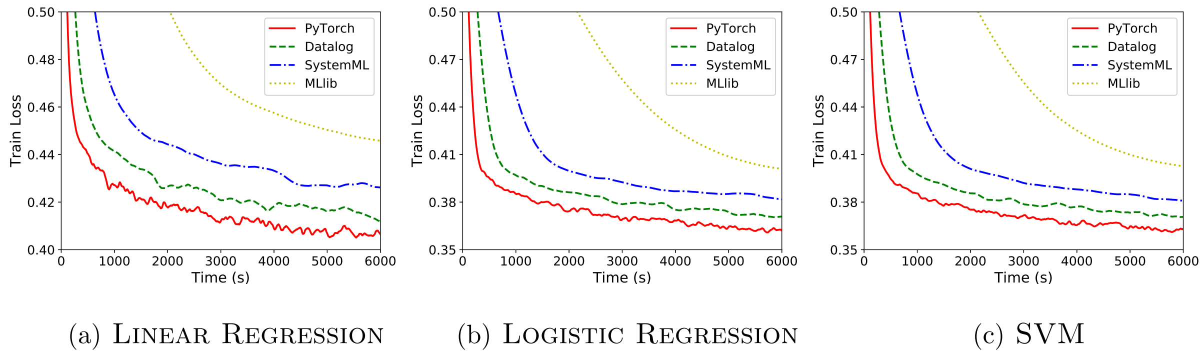

Figure 3.11: Performance Comparison: Training with Batch Gradient Descent

#### 3.8.1.3 Environment

The experiments of all the four systems are conducted on a cluster with 16 node: one node acts as the master and other 15 nodes as workers. For the distributed computing, since our *Datalog* framework, SystemML and MLlib are all based on Apache Spark, they use the bulk synchronous parallel architecture. Meanwhile, PyTorch runs under the parameter server architecture. All nodes are connected with 1Gbit network. 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. Each worker node is allocated 30 GB RAM and 8 CPU cores (120 total cores) for execution. BigDatalog is built on top of Spark 2.0 and Hadoop 2.2. All systems are activated with in-memory computation by default. Since hype-parameter tuning is outside the scope of this paper, the hyper-parameter settings are the same for all systems: the learning rate is $10^{-2}$ and the number of iterations for BGD is 100.

### 3.8.2 End-to-end Performance

To begin with, we report the end-to-end execution time of the three models trained with BGD. The results are shown in Figure 3.11, where our approach is denoted as *Datalog*. Note that some results of SystemML and PyTorch are denoted by the word "OOM" in red, since they run out of memory under those settings. We can make the following observations:




(a) Linear Regression (b) Logistic Regression (c) SVM

Figure 3.12: Performance Comparison: Training with Mini-batch Gradient Descent

Firstly, *Datalog* consistently outperforms the other two Spark based systems MLlib and SystemML for all three models. SystemML has the worst performance as its optimizations focus on physical-level computation within one iteration rather than the whole iterative training process. Such results make sense since the strong point of SystemML lies in directly computing the ML models by matrix operations. MLlib outperforms SystemML because it adopts a tree aggregate mechanism to accelerate the gradient computation in distributed environment; however *Datalog* is approximately 2X to 4X faster than MLlib. Our preliminary investigations suggest that performance gains of our approach over MLlib come from higher-level logical optimizations, which were particularly successful in reducing shuffle operations.

Secondly, the performance of *Datalog* is comparable with that of PyTorch, one of the most popular special-purposed ML systems. On some datasets, such as KDD10 dataset, *Datalog* even outperforms PyTorch by up to 2 times. This must be credited to our system's success in optimizing each computation step from planning to execution to fully harness the potential of Spark engine. We also see that PyTorch requires much more memory: it runs out of memory on the large datasets KDD12 and Webspam. A possible reason for that is that PyTorch needs additional memory to make a replica of gradients and parameters for each thread rather than each node. For large sparse dataset, PyTorch will run out of memory when broadcasting after an iteration.

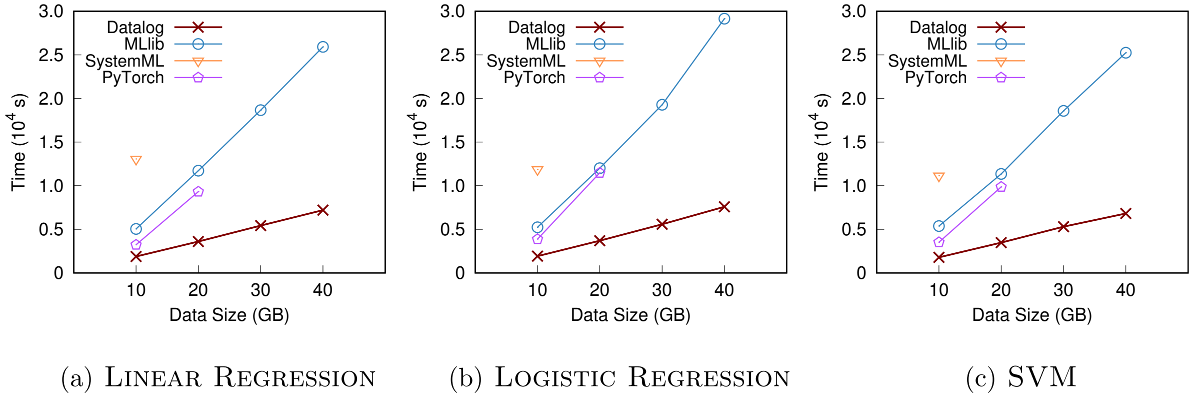
Lastly, the advantage of *Datalog* over other competitors is more obvious on larger datasets.

On the smallest dataset URL, the performance is comparable for all four systems. When it scales up to KDD10, MLlib and SystemML are approximately 2X and 5X slower than *Datalog*, respectively. For example, on the KDD10 dataset, the total execution time for LINEAR REGRESSIONon PyTorch, MLlib, and SystemML is 3889, 4689, 11351 seconds, respectively. While *Datalog* only takes 2338 seconds. Finally for KDD12, SystemML runs out of memory and *Datalog* outperforms MLlib by 5X. The possible reason for which SystemML runs out of memory could be that it conducts the ML application in the way in which matrix operations are optimized. Thus, even for sparse datasets, SystemML requires large volumes of memory to keep the intermediate results.

### 3.8.3 Results for Mini-batch GD

Next, we report the experimental results on training the three ML models with the MGD optimizer. We set the batch size as 8,192 empirically. Due to space limitations, we only report the results on KDD10 dataset. On the other datasets without memory issues, the results have similar trends. For experiments with MGD, we do not fix the number of iterations. Instead, the training process will terminate when convergence is reached (when the difference of training losses between two adjacent iterations is smaller than $10^{-3}$ or reaches the maximum 25,000 iterations).

As we can see from Figure 3.12, PyTorch has the best performance under most settings. This is not surprising since specialized ML systems have implemented several optimizations and improvements designed specifically for training with MGD. As it has been widely shown in previous studies, BGD is more suitable for ML systems based on relational engines, e.g. Spark and relational DBMS. Note that the main contribution claimed in this paper is to propose a purely declarative ML framework by taking advantage of the characteristics of Datalog, rather than implementing an ML system that provides richer and more efficient ML functions than other systems. Consequently, the main purpose of evaluation is to show that with the aggregates-in-recursion mechanism supported by sound optimization techniques, the




(a) Linear Regression (b) Logistic Regression (c) SVM

Figure 3.13: Scalability: Varying Data Size

ML workloads can be expressed by Datalog and its implementation can outperform other Spark based systems. Remarkably, our implementation of MGD with trade-off did show very promising results in the quality of training. The training loss that *Datalog* achieves at convergence for Linear Regression, Logistic Regressionand SVM is 0.418, 0.372 and 0.376, respectively; while that of PyTorch is 0.407, 0.363 and 0.365, respectively.

Moreover, we can see that *Datalog* converges faster than the other two Spark-based competitors while achieving similar training loss as PyTorch. For example, for the SVM model, *Datalog* requires only about 5,000 iterations to converge with 530 ms per iteration. Meanwhile, the results for SystemML is about 6,000 iterations with 1,048 ms per each iteration. Finally, MLlib had not reached converge after 20,000 seconds, which is beyond the x-axis of Figure 3.12.

### 3.8.4 Scalability

In a final set of experiments, we test the performance of BGD on different systems when scaling up the size of the training data. For that we used the synthetic datasets proposed in the previous study [TK18]. We vary the size of the dataset from 10GB to 40GB. Other detailed settings of the synthetic data are the same as that discussed in Section 3.5. Using the charts shown in Figure 3.13, we discover that *Datalog* achieves nearly linear scalability

for all three ML algorithms trained with BGD. This demonstrates the great potential of applying our approach to the workloads generated by larger training datasets.

Furthermore, we can also observe that *Datalog* consistently outperforms MLlib and SystemML for increasing cardinalities of the training sets. For example, for the Linear Regression model, *Datalog* outperforms MLlib by 2X to 6X and outperforms SystemML by up to one order of magnitude. Note that when the size of the dataset exceeds 20GB, PyTorch and SystemML run out of memory. Thus many data points are missing for these systems in the figures. This further demonstrates the advantage of our framework over other Spark-based ML systems. Moreover, our *Datalog* also achieves comparable performance with the special-purposed ML system PyTorch in scalability.

### 3.8.5 Results on Dense Datasets

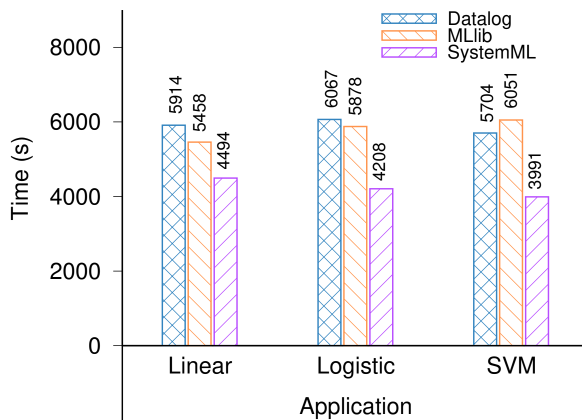



Figure 3.14: Performance Comparison on Dense Dataset

To include the whole spectrum of datasets and make a comprehensive evaluation, we also conduct experiments on a dense synthetic dataset. We continue using the synthetic datasets proposed in [TK18] but set the density as 0.5. We set the cardinality of dataset as 30 GB to make sure that all systems will not run out of memory [7]. There is no doubt that PyTorch

[7]Note that in previous experiments with sparse dataset, SystemML will run out of memory as it needs to

has much better performance than *Datalog*, MLlib and SystemML on dense data since it is optimized for supporting deep learning models, which involve many computations between dense matrices. In general, PyTorch could outperform Spark based system by an order of magnitude on such dense datasets. Therefore, here we only show the results of comparing with the other two Spark based systems SystemML and MLlib.

The results are shown in Figure 3.14. We can see that *Datalog* still achieves comparable performance with SystemML and MLlib. Although our proposed framework is designed for applications with sparse vectors, it still has reasonable performance on dense ones. It demonstrates the potential ability to extend the proposed techniques in *Datalog* to applications with dense training data.

## 3.9 Related Work

### 3.9.1 Datalog for Machine Learning

There has been some previous efforts in expressing ML applications with Datalog. Borkar et al. [BBC12a] came up with the proposal of a declarative workflow system, which also supports ML functionalities. Bu et al. [BBC12b] developed a Datalog query interface for it. MLog [LCC17] provided a set of imperative Datalog-style ML library over the TensorFlow system. LogiQL [MVP18] proposes to express ML applications with Datalog and script-like constructs. These studies focusing on using Datalog as part of the query interface. The work describe in this paper addresses the whole spectrum of advances needed to support effectively ML applications in Datalog and declarative query languages such as SQL.These include (i) formal declarative semantics for the query language, (ii) efficient system implementations with very effective optimization on parallel platforms, and (iii) considerable usability and interoperability in a data frame environment.

---

convert the dataset into its own data format, which would be much larger than the original sparse dataset as it might add some information to complement the omitted zero-dimensions

### 3.9.2 Recursive Query Processing

There is a long stream of research work about recursive query processing in the database community, seeking to provide formal declarative semantics for the usage of aggregates in recursion. Past research work tried to reach this goal by providing formal semantics for recursive Datalog programs with unstratified aggregates[GGZ91, MPR90, FGG02]. In particular, Ross et al. [RS92] used semantics based on specialized lattices to express the four aggregates, while Ganguly et al. [GGZ95] sought to optimize programs with extrema. More recently, Mazuran et al. [MSZ13] showed that continuous count and sum, are monotonic, and thus can be used freely in recursion. Monotonic aggregates have been implemented in the datalog system named DeALS [SYZ15] and scaled up to distributed systems [SYI16] and multi-core machines [YSZ17]. Recently, [ZYD17] introduced the Pre-mappability(PreM) property under which programs using min and max in recursion are equivalent to those aggregate-stratified programs. The extension of SQL with extrema in recursion [GWM19] is also realized based on PreM and has proved quite effective on graph applications. There is also new opportunities for reducing staleness and communication costs in distributed data processing [DZ19].

Past work also has recognized that Datalog is well-suited for large-scale analytical queries due to its amenability to data parallelism and the great expressive power of its recursive queries. In fact, Generalized Pivoting [SL91] and Parallel Semi-naive [SKH12] techniques enable parallel evaluation of *Datalog* programs. OverLog [LCH05] and NDlog [LCG06] proved effective at providing declarative networking. Systems that use Datalog to support data analytics in distributed environments include: SociaLite [SGL13], LogicBlox [ACG15], Myria [WBH15] and GraphRex [ZAC19]. However, the challenges of ML applications where not tackled by these systems. Therefore, they cannot support the queries expressed in this paper.

### 3.9.3 Large-scale Machine Learning

Supporting large-scale machine learning applications has become a hot topic in the database community. Several research works aim at optimizing the performance of linear algebra, which provides a common formal representation language for machine learning algorithms [TK18, EBH16, CKN17, ELB17]. Many previous studies focus on in-database machine learning. The basic idea is to formalize ML operators as optimization primitives and devise an engine on top of relational DBMS to solve the ML problem using them [FKR12, SOC16]. SimSQL [CVP13] employs a hybrid imperative and declarative framework to express linear models [LGG17], Bayesian learning [GLP17] as well as deep neural networks [JLY19]. While most previous solutions require many additional primitives, our framework is a purely declarative one that can be realized using basic constructs of Datalog or a simple relaxation of current SQL standards.

To take advantage of distributed data platforms, many ML frameworks were developed over Apache Spark as extensions. MLBase [KTD13] proposes a declarative ML framework by providing APIs of high level programming languages. Anderson et al. [ASS17] integrates Spark with MPI to improve the performance of graph and ML applications. KeystoneML [SVK17] and Helix [XMM18] provide more effective pipelines for ML workload. ML4all [KQT17] optimizes computation of gradient descent algorithms. PS2 [ZAC19] integrates the parameter server with Apache Spark. Our work shows that the advanced functionalities provided by such works can be expressed efficiently via Datalog supported by generalizing the existing query optimization and data parallelism techniques.

### 3.9.4 Machine Learning and Big Data Systems

Apache Spark [ZCD12] has been one of the most popular distributed data processing platforms which provides APIs for relational queries, graph analytics, data streaming and machine learning. DryadLINQ [YIF08], REX [MIG12] and Naiad [MMI13] provide effec-

tive interfaces to support large-scale workloads with iterations. Distributed graph systems provide vertex-centric APIs for graph analytics workloads. Typical examples include Graphlab [LGK12], Pregel [MAB10] and GraphX [GXD14].

Recently many ML systems have emerged to efficiently support different kinds of ML algorithms in distributed environment. The parameter server architecture [LAP14] opens up a new pathway to distributed model training. Examples adopting parameter servers include PyTorch [SDC19], TensorFlow [ABC16], Petuum [XHD15] and MXNet [CLL15]. SystemML [BDE16] is a declarative ML framework with plan optimizations on top of Apache Spark. Ray [MNW18] provides a unified interface that supports multiple tasks and settings.

## 3.10 Conclusion of Chapter

This paper has presented a powerful, declarative ML framework on top of Apache Spark based on Datalog. Thanks to the great expressive power of Datalog, users can write queries to express a series of ML algorithms trained by gradient descent optimizers without involving new constructs. The power of allowing aggregates in recursive Datalog programs is illustrated by the fact that it can be used for both expressing ad-hoc queries, and for producing a library of ML functions, i.e., a task for which procedural languages are normally required. We formally demonstrated that the training process expressed with Datalog programs has formal semantics by showing the Pre-Countable Cardinality property. Then, we proposed several planning and optimization techniques to efficiently support the evaluation of Datalog programs with complex recursions, which are essential to support ML applications. We also provided an equivalent SQL-based implementation with a very succinct syntax based on current SQL standard. Experiments on large-scale real world benchmarks demonstrated the superiority of our proposed framework over existing ML systems.

# CHAPTER 4

# Semantics of Completed Aggregates in Recursion

## 4.1 Introduction

The increasing requirements of Big Data applications have brought new challenges to Datalog researchers. Many efforts have been paid to combine the expressive power of recursive Prolog programs with the performance and scalability of relational DBMSs. Typical examples include the first commercial Datalog system LogicBox [ACG15] and the introduction of recursive queries stratified w.r.t. negation and aggregates into SQL-2003 standards. Recently there is a boosting demand for complicated analytical queries in many applications, such as graph search, machine learning and data mining. Such applications call for higher level of expressive power, scalability and performance, which brings new challenges for current DBMS and Datalog systems. While relational DBMSes gained great benefits from their success with descriptive analytics which can be supported via simple extensions of SQL-2 aggregates [CD97], they encountered major challenges for predictive analytics applications on Big Data. As is pointed out in [SAD10], MapReduce success is largely due to its ability of using relational DBMS data parallelism on new applications, e.g. Page Rank. Indeed, it has become a common view that SQL, the query language for relational DBMS, cannot be easily extended to such new application scenarios. As a result, researchers from both academic and industry fields focus on developing new special-purposed system to support these applications [YBT17]. For example, more than a dozen of graph database systems have been developed for this important application domain that is problematic for relational DBMSes.

Considering the many factors that led to this non-optimal situation, we observe that unresolved research issues played the most significant role in it. For instance, researchers have been aware of the fact that many algorithms can be expressed quite naturally in Datalog once aggregates and non-monotonic constructs are allowed to be used in recursions. There have been a long stream of research works towards this direction [MPR90, GGZ91, GGZ95, FGG02]. However, these early proposals suffer from many limitations, and non-monotonic reasoning research was still evolving discouraged premature commitments to a particular solution. Recent theoretical advances have witnessed major progress on the *stable-model* semantics. This is due to its great power and generality that extends beyond the original focus of Datalog to cover disjunctive programs and answer-set semantics. Unfortunately, its semantics require computational complexity levels that are unsuitable for the Big Data applications that have become the computer science cynosure. To address such issues, there is an overwhelming need for approaches with non-monotonic semantics that enables efficient implementations for a very wide range of applications.

An essential step towards the solution is to support the use of aggregates in recursion to express more advanced applications. Previous works have shown various ways in which aggregates can be used in recursive logic programs while retaining formal semantics. Thus, using aggregates that are monotonic in the lattice of set containment was discussed in [RS92]. Among the aggregates, using `min` and `max` in recursive programs that are equivalent to stratified programs was discussed in [ZYD17] and the PreM property was proposed. In this paper, we explore a third important situation where programs using non-monotonic aggregates nevertheless define monotonic mappings because those aggregates are applied to sets of known cardinality. In fact, the computation of an aggregate such as sum is performed in two phases. In the initial phase, we progressively add to the current continuous sum each item in the set. In the final phase, we detect the end of the input and return the last value produced in the initial phase. The desirability of clearly distinguishing between the two phases is well-recognized when dealing with continuous queries on data streams: in

fact, aggregates returning only the results from initial phase computation are non-blocking, whereas those returning results produced in the second phase are blocking. Likewise, the need to provide users with continuous aggregates that only compute in their initial phase was recognized in SQL:2003 with the introduction of OLAP Functions that support continuous aggregates. It is also very important to draw a clear distinction between the continuous and final version of the aggregate produced in the two phases when using the `min` recursion. Indeed, the continuous initial aggregates are monotonic,whereas the final ones are non-monotonic and they cannot be used as such in recursion. However,when the cardinality of the set is known, the the final phase computation can be recast in monotonic terms, whereby the whole aggregate becomes monotonic and can be used to express concisely and efficiently powerful recursive queries. We will use a series of running examples covering many typical application scenarios to illustrate it in this chapter later.

The rest of this chapter is organized as following: We introduce the formal definition of the set aggregation semantics in Section 4.2. We propose the PCC property and show how it provides the formal semantics for a full set of aggregates in Section 4.3. We demonstrate the formal semantics of queries for ML applications in Section 4.4. Finally, this chapter is concluded in Section 4.5.

## 4.2 Set Aggregation Semantics

In this section, we introduce the basic semantics of set aggregations, which serves as a foundation to illustrate the PCC property. We first present the basic definition of continuous count in Section 4.2.1 and then generalize it to sum and average aggregates in Section 4.2.2. Finally, we extend the discussion to group by aggregates in Section 4.2.3.

### 4.2.1 Basic Definition of Continuous Count

First we focus on the four basic aggregates, i.e.`count, min, max`and `avg` and define their formal declarative logic-base semantics and discuss effective operational approaches to realize them.

Suppose $\gamma$ denotes a set of distinct atoms; $p(X)$ denotes *continuous count* aggregate on $\gamma$ which returns the set of positive integers that do not exceeds the cardinality of the set. $\gamma$ and $p(X)$ can thus be computed using the Horn clauses in Definition 4.

**Definition 4 (Defining continuous count)** *The continuous count aggregate can be defined with the relation* `ccp` *generated with the following rules:*

$$
\begin{array}{llcl}
r_1 & ccp(C,[X]) & \leftarrow & p(X), C = 1. \\
r_2 & ccp(C1,[X|S]) & \leftarrow & p(X), ccp(C,L), C1 = C+1, new(X,L). \\
r_3 & new(X,[Y|L]) & \leftarrow & X <> Y, new(X,L). \\
r_4 & new(X,[\,]). & &
\end{array}
$$

Thus the goal `?ccp(X)` will progressively return integers up the actual cardinality of the above set $\gamma$. The name *monotonic count* is also used for continuous count, since it is defined using by Horn clauses that always generate monotonic mappings in the lattice of set-containment.

In terms of implementation, the above formal definition of monotonic count is quite inefficient since it constructs all possible permutations of the $X$ values, while only one of such permutations needs to be considered. Thus, actual realizations of continuous count in systems visit each atom in some efficient way—typically in the sequential order in which the atoms are stored.

The traditional final count used in SQL-2, i.e. the cardinality of set $S$, can be derived as the maximum of continuous count `ccp`. But rather than using the approach which defines

one aggregate using another, we can define it by the rules in Definition 4 and the following final rule:

$$r_5 \quad \mathit{final_count}(C) \quad \leftarrow \quad ccp(C, _), C1 = C + 1, \neg ccp(C1, _).$$

Unlike continuous count, the final count is defined using negation, which makes it non-monotonic. It is also observed that if our definition is applied to a set $S$ of infinite cardinality, it returns no result. Indeed, the set of natural numbers has no max, and contains no integer $C$ without a successor $C1$. Therefore in the following discussion, we will only consider aggregates computed on sets of finite cardinality. But as in the case of continuous count, its implementation will be expedited by considering only one of the possible permutations of the values in $S$. Then, since the system visits atoms in a particular order, the occurrence of the *completion condition* expressed by rule $r_5$ can be implemented by any test that determines whether it is the last atom in the set. For instance, if the $p(X)$ facts are stored in a file, then the stop condition is recognized by detecting that the next datum is the end-of-file (`EoF`) mark. This is just one way in which the stop condition is detected in systems. For instance, if the relation is indexed using a B+ tree, then the stop condition is detected by the fact we have completed the visiting of index block which has a null pointer to the next block. Moreover, if the stop condition is computed by a join or other relational algebra expressions, the stop condition is implied by the completion of the computation. Thus while the stop condition used in the computation of aggregates can be given a formal logic-based definition, its operational realization can be quite different according to the settings of different systems.

### 4.2.2 Extension to Sum and Average

The definition of other aggregates such as `sum` or `average` will use the template established for `count` consisting of an initial phase where their continuous version is computed, and then of a second phase where the final result is returned. Moreover, the final count can be used as the completion test that brings about the final phase in the computation of these aggregates.

**Definition 5 (continuous and final sum)** *The continuous count aggregate can be defined with the relation* `csc` *generated with the following rules:*

$$
\begin{array}{llcl}
r_6 & csc(S, C, [X]) & \leftarrow & p(X), S = X, C = 1. \\
r_7 & csc(S1, C1, [X|S]) & \leftarrow & p(X), csc(S, C, L), S1 = S + X, C1 = C + 1, \neg new(X, L). \\
r_8 & \mathit{final_sum}(S) & \leftarrow & csc(S, C, _), C1 = C + 1, \neg csc(C1, __).
\end{array}
$$

For example, the sum of $X$ values s.t. $p(X) \in S$ can be defined as Definition 5. Here rules $r_6$ and $r_7$ compute both the continuous sum and the continuous count as the first and the second arguments of `csc`. The value of final sum is actually that of the continuous sum when the continuous count value reaches the cardinality of $S$, i.e. a value that is equal to the final count. $r_8$ expresses this completion condition using the predicate `new` defined in Definition 4 for the computation of final count.

Thus, the sum aggregates is basically defined by a monotonic computation, except for a final rule that call on a non-monotonic final-count predicate. However, in many situations final count is known before we enter the recursive computation of `csc`. For instance, this is true when $S$ represents the atoms of a fixed-length vector. Moreover, in many situation where the cardinality of $S$ is not known, it can be actually computed using the program in Definition 4 to produce $final_count(C)$ in a lower stratum. Then rules $r_6$ and $r_7$ will still be used to compute cscm. But instead of $r_8$, the value of $final_sum$ will be computed with the rule $r_9$:

$$r_9 \quad \mathit{final_sum}(S) \quad \leftarrow \quad csc(S, C, _), final_count(C).$$

Therefore, the final sum aggregate can be implemented by a stratified program where the lower stratum perform the non-monotonic computation of final count, and the next stratum derives `sum` by a monotonic computation using rules $r_6$, $r_7$ and $r_9$ without negation. Thus, in our definition of `sum` we have combined the computation of `sum` and `count` into one stratified programs, where rules $r_1$ and $r_2$ occupy a lower stratum; $r_3$ containing negation is at higher stratum; and rules $r_4$, $r_5$ and $r_6$ are in a still higher stratum. Only $r_3$ that determines the actual count is non-monotonic.

In every program, recursive or not, when we compute `sum` on a set whose cardinality is known, $r_3$ is no longer needed and the computation of our aggregate becomes yet another monotonic predicated defined using Horn clauses. Similar observations also hold true for other aggregates such as average, and extrema aggregates. In fact, the `average` aggregate can be computed by replacing rule $r_8$ with:

$$final_avg(Avg) \quad \leftarrow \quad csp(S, C, _), Avg = S/C, final_count(C).$$

For `max`, we can instead write rules in Definition 4 where we use the predicate `larger` to return the larger of two values $M$ and $X$ (they cannot be equal since we are using set semantics).

**Example 4 (Defining the max on a set where final_count is known.)**

$$\begin{array}{rcl} ccs(S, C, [X]) & \leftarrow & p(X), M = X, C = 1. \\ cmp(S1, M1, [X|S]) & \leftarrow & p(X), cmp(M, C, L), larger(M, X, M1), notin(X, L). \\ larger(X, Y, X) & \leftarrow & X > Y. \\ larger(X, Y, Y) & \leftarrow & X \leq Y. \\ final_max(M) & \leftarrow & cmp(M, C, _), final_count(C). \end{array}$$

Dual definition holds for `min`, where instead of `smaller` we will use a predicate that returns the smaller of two values.

### 4.2.3 Group By Aggregates

The logical definition of aggregates specified with a group-by clause can be derived as an extension of above situations. Take the following rule for instance:

$$r_a \quad qs(X, sum\langle Y\rangle) \quad \leftarrow \quad pair(X, Y).$$

The joint computation of sum and count can be performed as Example 5.

**Example 5 (Defining continuous sum and final sum in the presence of group-by)**

$$
\begin{array}{llcl}
r_b & gbsc(X, S, C, [\,]) & \leftarrow & pair(X, Y), S = 0, C = 0. \\
r_c & gbsc(X, S1, C1, [[X, Y]|L]) & \leftarrow & pair(X, Y), gbsc(X, S, C, L), S1{=}S + Y, C1 = C + 1, \\
 & & & new([X, Y], L). \\
r_d & gbsc(X, S, C1, [[X, Y]|L]) & \leftarrow & pair(X1, Y), gbsc(X, S, C, L), X{<>}X1, C1{=}C + 1, \\
 & & & new([X1, Y]), L). \\
r_e & final_sum(S) & \leftarrow & gbsc(S, C, _), C1{=}C + 1, \neg gbsc(C1, _, _).
\end{array}
$$

Thus, the computation starts with $r_a$ to set the value of $S$ (sum) and $C$ (count) to zero. Then, after checking that the pair $[X, Y]$ is in fact new, in $r_c$ it increases the values of both $S$ and $C$ for the group-by values matching $X$, but in $r_d$ we only increase the $C$ value for the others. Thus at the end of the fixpoint computation, for each $X$ we will have the sum $S$ of the $Y$ values associated with it. The $C$ values will be the same for every group-by $X$ and equal to the cardinality of the set containing the $pair(X, Y)$ facts.

The last rule $r_e$ returns the final value of continuous sum when the continuous count has reached its final value. This is the only rule using negation. If we can replace it by the pre-computed cardinality, the whole computation of final sum becomes monotonic. Thus, as in the case where we had no group-by the Pre-Countable Cardinality of the sets involved assures that aggregates can be freely used in recursive definition. Furthermore, this conclusion also holds for count, avg, min and max, which can be reasoned in a similar way.

## 4.3 The Pre-Countable Cardinality Property

In this section, we formally introduce the Pre-Countable Cardinality Property. We first show the necessary background in Section 4.3.1. Next we illustrate the semantics provided by it in Section 4.3.2. Finally we use several examples to explain the usage of the property in Section 4.3.3.

### 4.3.1 Background

The semantic issues caused by the use of `count`, `sum`, and `average` aggregates in recursive computations require solutions that are different from those used for extrema due to their non-monotonic nature. For instance, the computation of `average` consists of two phases: in the first phase, monotonic rules are used to compute a pair $\langle num, total \rangle$ by increasing the $num$ by 1 (as in continuous count) and adding the new value (as in continuous sum). This monotonic phase completes when all elements in the set have been processed. In the second phase, the maximum value of $num$ and the value of $total$ associated with it are extracted. The ratio of the latter over the former is returned as the answer. This phase becomes non-monotonic due to the `max` aggregate used to obtain the maximum value of $num$.

The solution to this problem is based on the observation that set aggregates can be used in recursion when the cardinality of the involved relations can be pre-computed before entering the recursive computation, and simply passed to the fixpoint computation that follows. In above example, if the cardinality of a relation was pre-computed, then we can simply select that value and the $total$ value associated with it to get the answer, thus eliminating the `max` aggregate which is the only non-monotonic construct involved in the computation. Then the `average` aggregate expressed using monotonic constructs can be used freely in recursion. Moreover, to compute the `sum` we can still compute the pair $\langle num, total \rangle$ in order to achieve monotonicity, but then only return the value of $total$ as the result. Remarkably, this *Pre-Countable Cardinality* (PCC) condition occurs in many programs of great practical significance of Datalog. We will now formally provide the PCC idea in Definition 6.

**Definition 6 (PCC)** *Let $R$ be a recursive relation in Datalog, and $\delta R_i$ denotes delta values of $R$ obtained at each iteration $i$ during the SN fixpoint computation. Then $R$ satisfies the* PCC *condition when: (i) The cardinality of $\delta R_i$ is non-zero and is the same for each $i$; (ii) The cardinality of $\delta R_i$ can be determined before the SN fixpoint computation begins.*

### 4.3.2 Semantics provided by PCC

As previously described, the non-monotonic aggregate `sum` can be computed by incrementally computing the pair $\langle num, total \rangle$ and returning the *total* value associated with the *num* value *that is equal to the cardinality pre-computable before the recursive computation*. In this way, the computation process will involve only monotonic constructs, since the incremental computation of continuous count and sum is monotonic. In other words, the program with `sum` aggregates in recursion is equivalent to stratified programs where the cardinality is pre-computable at a lower stratum, which precedes the SN computation of the equivalent program that only use monotonic constructs [1] Observing that similar properties also hold for other aggregates, we can summarize our finding in Theorem 1.

**Theorem 1** *If the* PCC *property is satisfied by a recursive Datalog program $P$ that uses* ***sum****,* ***avg*** *and* ***count*** *in recursion, then there exists an equivalent aggregate-stratified program of $P$, which defines its formal semantics.*

### 4.3.3 Examples

Next we provide two examples to show how PCC can be used in real world applications.

**Example 6 (The Markov Chain algorithm)**

$$
\begin{aligned}
next(0, C, sum\langle In \rangle) &\leftarrow mov(C, C, _), In = InitPop.\\
next(J1, To, sum\langle In \rangle) &\leftarrow next(J, C, Pop), mov(C, To, Perc),\\
&\quad\; In = Pop{\times}Perc, J1{=}J{+}1), J1 \leq 1000,\\
&\quad\; JL = J - 1, next(JL, C, PopL), PopL - Pop > 0.\\
finalstep(max\langle J \rangle) &\leftarrow next(J, _, _).\\
fpop(C, Pop) &\leftarrow finalstep(J), next(J, C, Pop).
\end{aligned}
$$

---

[1] If this program contains min and max, a third stratum is needed on top of these two to defined its formal stratified semantics.

Example 6 shows a *Datalog* program to express the procedure of calculating the Markov Chain algorithm. Assume that the base relation is `mov(C, To, Perc)`, which respectively describe the names of cities of interest and the fraction of population that will move from `C` to `To` in the course of a year. For each city, there is also a non-zero arc from the city back to the same city showing people that will not move away. Therefore, the sum of `Perc` for arcs leaving the city (i.e., a node) is always equal to one. Thus, assuming that initially every city has a population `InitPop`, we need to find how the population evolves over the years.

In this example, the last two rules specify post-conditions that must applied at the end of fixpoint computation. Nevertheless, it is quite straightforward for the compiler to integrate them into the semi-naive fixpoint computation to achieve a significant optimization. In fact, during the semi-naive fixpoint computation, the delta relation can be automatically identified at the end of each iteration. Moreover, the latest delta atoms are identified by the largest value of `J`, which is also the max value of the index, i.e. the values returned by $r_e$ upon termination. Thus $r_e$ and $r_f$ can be implemented by simply returning the latest delta atoms upon termination. Now, since only atoms for the maximum value of $J$ are needed, all facts with other $J$ values can be dropped to achieve a much more efficient usage of memory.

In this program, it is obvious that PCC holds for the `sum` aggregates. Therefore, we have a formal semantics defined by a stratified program consisting of (i) a bottom stratum where count is defined; (ii) a middle stratum of Horn clauses, i.e., monotonic rules; and (iii) a top stratum used to post-select the final results of interest. Finally, this formal semantics are realized via a very efficient operational semantics that only requires the semi-naive computation in the middle stratum, inasmuch as the completion of join in (i) replaces the completion of the final count, and the extraction of the final results in (iii) is realized by the selection of the final delta in the semi-naive fixpoint.

**Example 7 (Clustering a lá Lloyd)**

$$r_a \quad center(0, Cno, Dim, Val) \quad \leftarrow \quad init(Cno, Dim, Val).$$

$$
\begin{aligned}
r_b \quad dist(J, Pno,\ Cno,\ sum\langle SqDis\rangle) &\leftarrow point(Pno, Dim, Val), center(J, Cno, Dim, CVal),\\
&\quad\ SqDis = (Val - Cval) * (Val - Cval).\\
r_c \quad mindist(J, Pno,\ min\langle DCno\rangle) &\leftarrow dist(J, Pno, Cno, DSm), encd(DSm, Cno, DCno).\\
r_d \quad center(J1, Cno, Dim, avg\langle Val\rangle) &\leftarrow mindist(J, Pno, DmCno), decd(DmCno, _, Cno),\\
&\quad\ points(Pno, Dim, Val), J1 = J + 1.
\end{aligned}
$$

Example 7 demonstrates the Lloyd's Algorithm for K-means Clustering. The base relation is a large set of $D$-dimensional points. Each point is described by a unique `Pno` and the coordinate value `Val` in each dimension denoted by `Dim`. We also have a small set of centroids. Then to generate the initial assignment `center(0, Cno, Dim, Val)`, we used the relation `init` that implements one of the many techniques described in the previous studies. At each step $J$, the algorithm finds the closest center for each point. Then a new set can be generated by averaging their coordinates.

We then show how this example satisfies the PCC property. If we let $|P|$, $|C|$ and $|D|$ denote the cardinalities of set of points, centers, and dimensions, respectively. We have that, for each rule, the aggregate computation involves a number of elements that is independent of $J$, since $r_b$ specifies a computation taking place over $|P| \times |C| \times |D|$; $r_c$ specifies a computation is over $|P| \times |C|$ elements; while the computation of $r_d$ takes place over $|P| \times |D|$ elements. These are counts that can be easily determined before the recursive computation and remain the same for every value of $J$. These explicit values could be passed to the recursive rules for computing the monotonic aggregates used in these rules. A much simpler and efficient solution consists in letting the system detect executions of the body operators at each step $J$, which has already been implemented as part of the optimized semi-naive fixpoint computation.

## 4.4 Formal Semantics of Machine Learning Applications

We find that ML applications tend to apply aggregates over sets of relations whose cardinality can be pre-computed ahead of time, where the computation of all kinds of aggregates becomes monotonic. Following this route, we can also provide a formal semantics for a wider spectrum of applications expressed in Datalog from the aspect of fixpoint computation. In this section, we thus show that the semi-naive fixpoint computation of Query 3 of Section 3.3 indeed realizes the formal semantics defined above. In fact, the first `J` in Query 2 coincides with the successive steps of the semi-naive fixpoint, and the cardinality of arguments of the `sum` aggregate remains the same at each step, and can in fact be pre-computed before the recursive computation starts. Here the value $n$ is the cardinality of training set, i.e. *vtrain*. In the process of recursive computation, a step of the semi-naive computation terminates after processing exactly the same number $n$ of input values for each value of `J`. Thus the SN computation for Query 2 realizes the formal fixpoint semantics of the equivalent stratified where the cardinality is pre-computed before the semi-naive fixpoint computation begins.

Then we formally conclude these findings with the following Theorem 2.

**Theorem 2** *The results of Query 2 are semantically equivalent to the same result with a stratified query that does not have aggregates in recursion.*

We can use the similar techniques proposed in [GWM19] for testing PreM to enable automatically testing of the PCC property.

## 4.5 Conclusion of Chapter

In this chapter, we investigate the formal semantics on a completed set of aggregates in recursive queries. We demonstrate that recursive computations on datasets of fixed cardinality represent an area of great theoretical and practical interest for Datalog and other

logic-based languages. Indeed, we have shown that many important analytical queries can be efficiently expressed using aggregates in recursion, while avoiding the difficult semantic issues besetting the use of non-monotonic constructs in recursive programs. We have found that, when the number of such facts in the world remains unchanged, aggregates on the attributes of them can be used in recursive logic rules while preserving the desirable properties of fixpoint computations. With such a theoretical tool, we provide the formal semantics of the queries designed for ML applications. This and other recent results using the PreM property of extrema suggest that aggregate can provide the long-sought bridge between formal non-monotonic semantics and efficient implementations that, over many years of work, could not be built by non-monotonic reasoning researchers using only negation.

# CHAPTER 5

# Optimizing Parallel Recursive Datalog Evaluation on Multi-core Machines

## 5.1 Introduction

In the past years, there is a resurgence of Datalog due to its ability to specify declarative data-intensive applications that execute efficiently over different systems and architectures. The recent theoretical advances [ZYD17, MSZ13] enable the usage of aggregates in recursions, and this leads to considerable improvements in the expressive power of Datalog. As a result, Datalog has been widely adopted to express complicated recursive queries in many domains, such as artificial intelligence [Dar20], graph analysis [ATO16], knowledge reasoning [BSG18], declarative network [LCG06] and many others.

With the ever-growing scale of data analysis tasks, a high level of performance and scalability becomes critical for Datalog systems. In response to this need, many parallel Datalog engines have been developed by researchers from both academia and industry. Based on the environment they are deployed, these Datalog engines can be divided into two categories: shared-memory [SGL13, YSZ17, FZZ19] and shared-nothing [SYI16, WBH15, SPS13] ones. These approaches implement the idea of parallel bottom-up evaluation [GST92] by splitting the tables into disjoint partitions via discriminating functions, such as hashing, where each partition is then mapped to one of the parallel workers. After each iteration, workers coordinate with each other to exchange newly generated tuples when necessary. The final result is the union of contributions by all workers. In this way, the entire computation can be divided

among all workers and operated in parallel.

Witnessing the emergence of modern commodity machines with massively parallel processors [AGN13], it is shown in previous studies that shared-memory multicore architectures have demonstrated superior performance for Datalog applications. However, these studies either (i) underutilize the multicore architecture due to poor parallelism [YSZ17]; or (ii) are based on different system architectures [FZZ19, SJS16, SGL13]. Therefore, while these studies provide highly valuable techniques, mechanisms and execution models, none of them uses the knowledge at hand to solve the problem we address here.

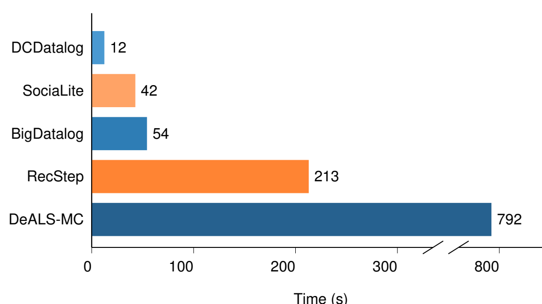



Figure 5.1: Query Performance of SSSP on LiveJournal Dataset

In this paper, we propose **D**ynamic **C**oordinate **Datalog** (DCDatalog), a parallel engine on shared-memory multicore machines to scale up Datalog evaluation. The key challenge of finding a good plan for parallel evaluating recursive Datalog programs is that it should provide an efficient mechanism to resolve the race conditions, which requires to ensure the atomicity of concurrent update operations. The existing study [YSZ17] solves this problem by just identifying a family of lock-free programs and forcing global coordination after each iteration in the parallel evaluation plan. Firstly, the scope of lock-free programs or those can be turned into lock-free ones by simple rules rewriting is rather narrow. As a result, it fails to support more complicated applications, especially those with aggregates in recursion. Secondly, it will incur costly coordination overhead and give rise to the problem of poor parallelism, where the faster workers are blocked by waiting for the slowest ones to arrive at the coordination point. Witnessing this problem, we propose a new system architecture to elim-

inate the limitation of lock-free or decomposable programs on Datalog applications as well as boost the performance in parallel evaluation. We deal with race conditions by leveraging a light-weight scheme that can get rid of the global coordination among workers. This is enabled by the newly proposed **D**ynamic **W**eight-based **S**trategy (DWS). Instead of blocking the faster workers, DWS reduces the straggler with effective local checkers controlled by a simple weight-based mechanism. Since such weights are calculated on-the-fly, the workers can dynamically make the decision about whether to perform idle waiting or proceed to the next iteration. This relaxation will definitely reduce the overall execution time and improve the parallelism of evaluation plans. For example, the performance of Single Source Shortest Path (SSSP) query depicted in Figure 5.1 shows that DCDatalog is considerably better than other competitors since it is equipped with a better coordination strategy. We conduct extensive experiments using five widely used recursive Datalog programs on several real world datasets. The experimental results demonstrate that our DCDatalog engine provides across-board performance gain and outperforms existing Datalog engines by one to two orders of magnitude.

The rest of this paper is organized as following: We provide some necessary backgrounds and introduce the overall system architecture in Section 5.2. We propose the dynamic coordination strategy in Section 5.3. We show the experimental results in Section 5.4. We survey the related work in Section 5.5. Finally the conclusion is made in Section 5.6.

## 5.2 Preliminary

### 5.2.1 Parallel Evaluation of Datalog Programs

There are two kinds of parallel execution architectures: shared-memory and shared-nothing. In the shared-memory architecture, all the base and recursive relations are stored in memory

that can be directly accessed by all processors [1], as supported by most modern multicore machines. When different processors visit the same piece of memory, the race condition happens once at least one of the operation is write. In this case, the lock mechanism is required to ensure the atomicity of operations. The examples of Datalog engines under this architecture include DeALS-MC [YSZ17], Souffle [JSZ19] and RecStep [FZZ19]. In this paper, we focus on this category of studies. In the shared-nothing architecture, the data is distributed into different computation nodes. The nodes in a cluster use the message passing mechanism to exchange information with each other, which involves extra network communication. The examples in shared-nothing architecture include BigDatalog [SYI16], Distributed SociaLite [SPS13] and Myria [WBH15].

The state-of-the-art method for evaluating Datalog programs in parallel is the *substitution partitioned parallelization* scheme [GST92]. It first divides the workload into $n$ disjoint partitions using hash-based discrimination function. Then each partition is assigned to exactly one worker. Since such partitions are disjoint with each other, each worker operates on a distinct non-overlapping partition during the bottom-up SN evaluation, which thus avoid redundant computation. The correctness of such a parallel execution can be summarized as Definition 7, which has been formally proved in [GST92].

**Definition 7** *Suppose $P$ is a recursive Datalog program to be executed over $n$ workers. Under above partitioned parallelization scheme, suppose $Q_i$ is the program to be executed at worker $i$ and $Q = \cup_{i=1}^{n} Q_i$. For every interpretation, the least model of the recursive relation in $Q$ is identical to the least fixpoint obtained from the sequential execution of $P$.*

To describe this process, we say that a *local iteration* is executed by a worker when it finishes one iteration of SN evaluation; while a *global iteration* is executed when all workers have finished the same number of local iteration. If the delta table becomes empty after a

[1] We will use the terms processor, thread and worker interchangeably if there is no ambiguity in the context.

local iteration, the *local fixpoint* is reached; the parallel evaluation terminates if the *global fixpoint* is reached.

Although the above method is theoretically sound, many issues must be addressed before it can be turned into practical system implementation. Specifically, we need to address the issue of how to deal with race conditions in parallel execution. Previously the DeALS-MC [YSZ17] system achieves parallelization for a set of programs that are either lock-free, or can be translated into lock-free programs via simple rule rewriting. However, the lock-free conditions required by DeALS-MC are too strict and many Datalog programs with aggregates in recursion, such as Connected Component, Single Source Shortest Path and Path Counting, do not satisfy such conditions. In order to achieve a correct evaluation for such programs, the programmers need to generate their execution plans manually by inserting locks into proper places that race conditions might happen. Furthermore, even for lock-free programs, coordination among all the workers must be enforced after each global iteration. This will cause the fast workers to be blocked since they cannot move on to the next local iteration until slow straddlers complete the current global iteration.

### 5.2.2 Overall Framework

In this paper, we aim at developing a Datalog engine which, by eliminating the requirement that programs must be lock-free, will support a wide scope of applications with superior performance and scalability.

To reach this goal, we propose a new architecture for parallel Datalog evaluation, based on the following key design principles about its runtime. Firstly, we subdivide the memory space into partitions with finer granularity. Then, rather than let workers request information from all others after a global iteration, each worker just sends the newly generated delta table to the memory space owned by other workers after its local iteration. In this way, we eliminate the requirement of global coordination, and thus significantly save the time of idle waiting. The dynamic strategy DWS used to control this process is described in Section 5.3. Secondly,

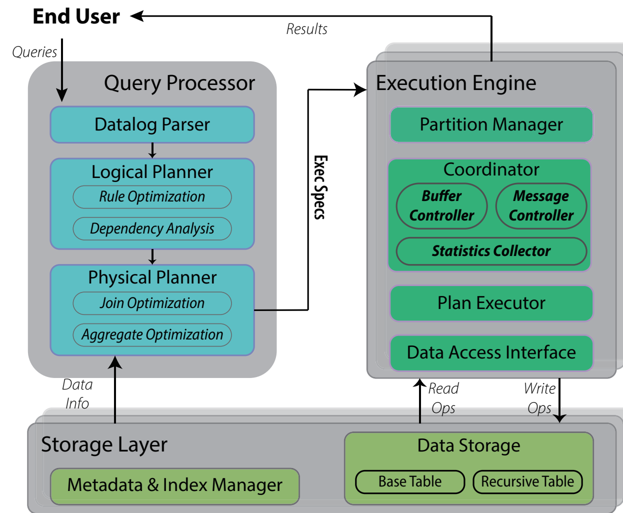



Figure 5.2: DCDatalog: The Overall Architecture

instead of locks, we use light-weight *atomic operations* to deal with race conditions among workers. Such an approach provides an efficient implementation of DWS and significantly reduces the overhead brought by locks.

The overall architecture of DCDatalog is shown in Figure 5.2. It consists of the following components:

**Query Processor** The Query Processor provides the functionality of analyzing and planning for input Datalog programs. It consists of three steps: (i) The *Datalog Parser* compiles the input Datalog program and generates its Predicated Connected Graph (PCG) [AOT03], which is implemented with the data structure of *AND/OR Tree*; (ii) The *Logical Planner* maps the AND/OR Tree into relational operators to form the logical plan; (iii) The *Physical Planner* further generates the physical plan to be executed in parallel.

**Execution Engine** The Execution Engine aims at providing efficient and scalable implementation to execute operators in the physical plan. The *Partition Manager* splits the input data into disjoint partitions with hashing mechanism[2]. The *Coordinator* implements

[2]The choice of hashing mechanism is beyond the scope of this paper. Here we just use a popular hash

the DWS strategy that controls execution at runtime. The *Buffer and Message Controller* is responsible for managing the memory access and information exchange among different workers. Various optimizations affecting the computation of aggregates in recursion are also performed by this component.

**Storage Layer** The Storage Layer provides the index and storage functions for the base and recursive relations during the SN evaluation process. In this paper, we utilize the storage engine of the DeALS system [SYZ15] along with a new the $B^+$-Tree index implemented by ourselves. Alternatively, other relational DBMSes could be also used as the storage and index engines.

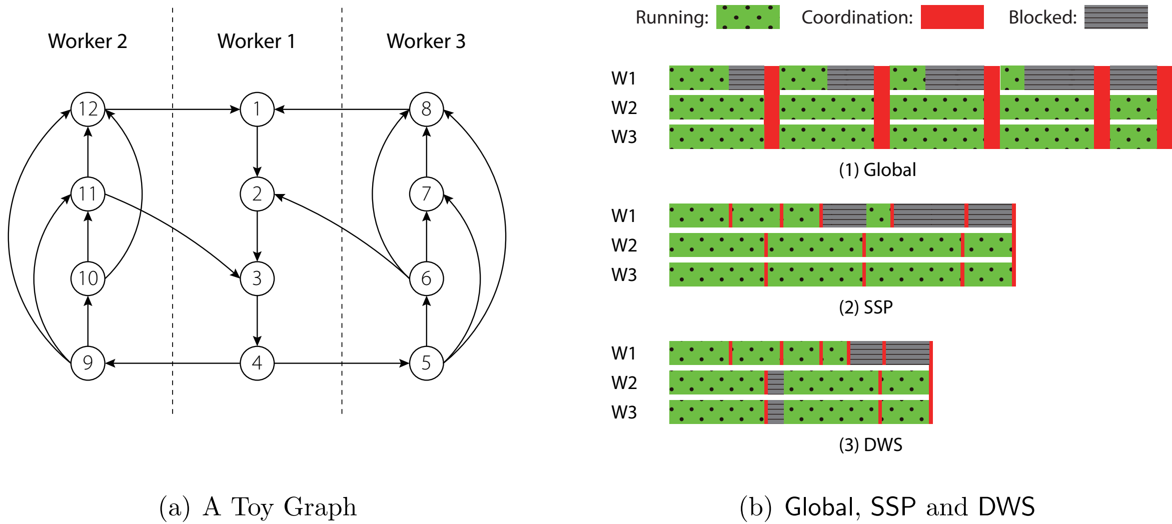
## 5.3 Dynamic Coordination Strategy

In this section, we present the dynamic coordination strategy used in the shared-memory multicore architecture. In Section 5.3.1 below, we introduce the basic parallel execution coordination process along with a straightforward improvement. Then in Section 5.3.2, we propose the new strategy DWS to optimize the performance. Finally, we provide a formal proof for its correctness in Section 5.3.3.

### 5.3.1 Parallel Execution Mechanism

We start from the general framework for parallel bottom-up evaluation described in Algorithm 3. The algorithm first splits the key range into disjoint partitions using the predefined hash function $H$ (line 2). Here we just follow the previous study [YSZ17] and use the same hash function to make partitions of both base and recursive tables. Suppose there are $m$ partitions $P_1, ..., P_m$ and $n$ workers $W_1, ..., W_n$ ($m \geq n$), all workers will run in parallel, while each deals with its own partition. Next, we build a hash index for each partition of the base

function for integer.




(a) A Toy Graph

(b) Global, SSP and DWS

Figure 5.3: Execution Time under Different Coordination Strategies

relation to accelerate the evaluation (line 3). Then all workers start with an active state and execute the SN evaluation in parallel: each worker $W_i$ will first initialize its recursive table $R_i$ according to the base rule (line 8) and build a B-Tree index on $R_i$ (line 9). Next, a local iteration of semi-naive evaluation is processed and the new delta relation $\delta R_i'$ is generated (line 11). After all workers finish a global iteration, they will coordinate with each other by exchanging the newly generated tuples according to $H$ and then proceed to the next iteration (line 13). We call this simple approach Global. At this time, we also need to update indices on recursive tables and perform deduplication. In this process, If $\delta R_i'$ is empty, then the local fixpoint has been reached, and thus $W_i$ is set to inactive (line 15). But if $\delta R_i$ of an inactive worker $W_i$ becomes non-empty after coordinating with other workers, it will become active again(line 17). The parallel evaluation terminates when all workers become inactive (line 20), denoting that the global fixpoint is reached. The final result is equal to the union of recursive tables of all workers (line 22).

*Query 6* - Connected Component (CC)

$$
\begin{aligned}
\mathsf{r}_{2,1}: \quad cc2(Y, min\langle Y\rangle) &\leftarrow arc(Y, _). \\
\mathsf{r}_{2,2}: \quad cc2(Y, min\langle Z\rangle) &\leftarrow cc2(X, Z), arc(X, Y). \\
\mathsf{r}_{2,3}: \quad cc(Y, min\langle Z\rangle) &\leftarrow cc2(Y, Z).
\end{aligned}
$$

**Example 8** *We evaluate the* CC *program on the graph shown in Figure 5.3(a). The execution process of* Global *(Algorithm 3) is displayed in Figure 5.3(b)(1). The workers $W_2$, $W_3$ are slower than $W_1$ since they are associated with more edges. In the first global iteration, workers $W_1$, $W_2$ and $W_3$ take 5,8,8 time units, respectively. Under* Global*, when $W_1$ finishes running, it is blocked as it should wait for other workers to finish the global iteration. Once $W_2$ and $W_3$ have finished their work, they coordinate and exchange their newly generated tuples with each other together with $W_1$. At that time, $W_2$ and $W_3$ are aware of the connected component with vertex 4. That is, they realize the connected component with vertex 1 after 4 global iterations. Finally,* Global *spends 128 time units in total.*

We can see that in above procedure idle waiting might happen before coordination (line 13). This is due to the requirement that all workers should wait until the current global iteration is finished. For complex queries that are not lock-free, coordination among all workers can result in serious race conditions, which will involve significant overhead. To address this problem, we first provide a straightforward improvement by extending the method of Datalog evaluation under shared-nothing settings recently proposed in [DZ19]. This method uses the Stale-Synchronous Parallel (SSP) model which was previously proposed for distributed machine learning [HCC13, CCH14]. The core idea in this approach is that we can relax the constraints now imposed on local iterations as follows: Instead of conducting just one local iteration and then waiting, we allow all workers to continue executing at most $s$ local iterations before stopping to wait for the current global iteration to be finished. Thus the intuition driving our approach consists in having workers spend more time performing actual computation rather than idle waiting for straddlers to finish. These benefits could be optimized by carefully tuning the hyper-parameter $s$.

**Algorithm 3**: Parallel Evaluation ($B$, $H$)

**Input**: $B$: The base table, $H$: The hash function for partition

**Output**: $\mathcal{R}$: All results in the recursive table

```
1  begin
2      Split the key range into disjoint partitions with H;
3      Construct Index for the each partition of B on the partition key;
4      while True do
5          foreach worker W_i do
6              // Run in parallel
7              if In the first iteration then
8                  Initialize R_i and δR_i with the base rule;
9                  Construct an index on R_i;
10             if W_i is active then
11                 Conduct one local iteration of evaluation with δR_i and generate δR'_i;
12             // Wait until global iteration is finished
13             Coordinate with all workers to update δR_i according to H, update the
               index for R_i;
14             if δR_i = ∅ then
15                 Mark W_i as inactive;
16             else
17                 Mark W_i as active;
18
19         if All workers are inactive then
20             Terminate the evaluation;
21
22     return R as the union of all workers;
23 end
```

**Example 9** *Let's look back to the previous example, suppose $s = 1$ in the SSP method. As depicted in Figure 5.3(b)(2), $W_1$ is not blocked by $W_2$ and $W_3$ in the first 3 local iterations as it can proceed one iteration ahead of them under SSP. After that, due to the constraint $s = 1$, $W_1$ needs to wait until $W_2$ and $W_3$ finish their $2^{nd}$ local iteration. In this example, the coordination only takes 1 time unit since it only requires the information dispatched by one worker. Even so, SSP finishes in 88 time units, i.e., about 40% faster than Global.*

To alleviate the overhead brought by race conditions, we split the main memory space into units with finer granularity: Each worker is associated with a segment of memory buffer $\mathcal{M}_i$ to hold the delta relation newly generated by other workers whose key falls in the range of $W_i$ under $H$. Specifically, the space for storing newly generated tuple from worker $W_j$ is denoted as $\mathsf{M}_i^j$. In this way, when performing coordination among workers, the race condition will happen just in a buffer $B_i$ rather than the whole memory space.

### 5.3.2 The DWS Approach

Although the SSP-based strategy is quite effective, some limitations still remain. In particular, since the value of $s$ remains unchanged, there is no guarantee that it will be the best for all workers during the whole evaluation process. Consequently, it is very difficult to set a proper value for $s$. Moreover, it still requires coordination after each global iteration, causing additional overhead due to race conditions.

Based on the above observation, we propose the **D**ynamic **W**eight-based **S**trategy (DWS) to further improve coordination during parallel SN evaluation. In DWS, we eliminate the requirement of global coordination. This is realized by allowing workers to automatically decide whether to proceed to the next iteration after a local iteration is finished. During the evaluation by a worker $W_i$, the evaluation time per iteration depends on the cardinality of its delta table. If this cardinality is rather small, it means that the worker $W_i$ is probably faster than other workers. In this case, $W_i$ should pause to collect more tuples from slower

workers. Otherwise, $W_i$ should allocate the newly generated tuples to the message buffers $\mathsf{M}_j^i$ of other workers $W_j$ ($j \neq i$) and move on to the next iteration. To this end, we need the following two parameters for each worker $i$: $\tau_i$ is the time $W_i$ should wait before proceeding to the next iteration; $\omega_i$ is a threshold such that $W_i$ will proceed to the next iteration if the cardinality of delta table is larger than it. Since the values of such parameters can be automatically calculated in each iteration, we can avoid manual tuning and take advantage of more background knowledge for coordination. Moreover, as each worker just updates the memory buffer with the delta table in an asynchronous manner, there will be fewer chances for race conditions on the memory buffer of each worker.

Algorithm 4 describes the behavior of DWS in each iteration. In a way similar to Algorithm 3, it first initializes the recursive table $R_i$ and the delta table $\delta R_i$ from base rules. If $W_i$ is active, it will collect newly generated tuples from the memory buffer $\mathsf{M}_i^j$, remove tuples already existed in $R_i$ and merge them into the delta table. (line 4). Then the algorithm makes a decision according to the cardinality of newly generated delta table. A cardinality smaller than $\omega_i$ means the algorithm must wait for $\tau_i$ time units during which it will collect more tuples from other workers before going back to computing $\delta R_i$ [3] (line 6). If the cardinality is zero, then the local fixpoint is reached and $W_i$ becomes inactive (line 8). Otherwise, the evaluation proceeds to the next iteration by (i) updating the two parameters with a weight-based mechanism (line 10), (ii) performing one iteration of evaluation with the collected delta table $\delta R_i$ (line 11), (iii) sending the tuples of newly generated delta table $\delta R_i'$ to buffers $\mathsf{M}_j^i$ of other workers and preparing for the next iteration (line 12-15).

[3]Starvation and Deadlock can be avoided in this process via common techniques in operating system such as setting a maximum total waiting time.

**Algorithm 4**: Execution of DWS on worker $W_i$

```
1  begin
2      //Replace line 10 to 18 in Algorithm 3, all workers are active in the beginning
3      if W_i is active then
4          δR_i ← (∪_j M_i^j − R_i) ∪ δR_i;
5      if 0 < |δR_i| < ω_i then
6          Let W_i wait for τ_i time units and then goto line 4;
7      else if |δR_i| = 0 then
8          Make W_i as inactive;
9      else
10         Update ω_i and τ_i;
11         Conduct one local iteration of evaluation with δR_i, generate δR'_i;
12         if ∃ tuple R ∈ δR'_i associated with W_j then
13             Update M_j^i and make W_j as active;
14         R_i ← R_i ∪ δR_i;
15         Update the index for R_i, δR_i ← ∅;
16
17 end
```

**Example 10** *As shown in Figure 5.3(b)(3), $W_1$ is never blocked thus it can quickly propagate the connected component with vertex 1 to other workers. Besides, to reduce the unnecessary computation happened in SSP, $W_2$ and $W_3$ wait for a short while to obtain the newly generated tuples produced by $W_1$ in its second local iteration. As a result of these improvements, DWS, with little additional waiting time included, requires 67 time units: this is about half the time of Global and represents a solid improvement over SSP.*

A remaining issue is to dynamically adjust the values of $\omega_i$ and $\tau_i$ for each worker $W_i$. We propose a weight-based mechanism to decide the values according to the statistical

information collected after each iteration. The reason for which $W_i$ waits for $\tau_i$ time units is to collect enough tuples for its delta table from slower workers. Therefore, the value of $\tau_i$ should be determined by all workers $W_j$ that have ever sent tuples into $\mathsf{M}_i^j$. Thus, we use the average of evaluation time spent by these workers in the last iteration as the value of $\tau_i$. Suppose there are $b$ workers $W_j$ s.t. $\mathsf{M}_i^j \neq \emptyset$, then we have: $\tau_i = \frac{1}{b} \sum_{j, \mathsf{M}_i^j \neq \emptyset} \tau_j$. To take the historical information into consideration, we further summarize the value of $\tau_i$ in all iterations till now by assigning more recent iterations higher weights. Specifically, in the $t$-th iteration, the value of $\tau_i^t$ is calculated as Equation (5.1):

$$\tau_i^t = \frac{1}{\sum_{k=1}^{t-1} e^{k-t}} \sum_{k=1}^{t-1} e^{k-t} * \tau_i^k \tag{5.1}$$

where $e^{k-t}$ is the weight for the $k$-th iteration.

Similarly, the value of $\omega_i$ should be determined by considering both the cardinality of the delta table generated in $W_i$ and the number of tuples that will be sent to buffers of other workers. To this end, we need to estimate the cardinality of the delta table generated in the $k$-th iteration, by using the historical information collected in the first $k-1$ iterations. Such estimation can be realized by maintaining a histogram on each worker, where the key range on a worker is subdivided into a set of buckets. After each iteration, the number of join results in each bucket is updated accordingly. When the delta table $\delta R_i$ is received by $W_i$, we can estimate the cardinality of join results by leveraging its histogram. Suppose we have $c$ buckets, and $\delta R_i$ contains $z_l$ records whose keys match the range of corresponding buckets in the histogram. The cardinality of newly generated tuples can be derived in linear time as $\sum_{l=1}^{c} z_l$. Next, we estimate the number of tuples that another worker $W_j$ expects to receive from $W_i$. As each worker $j$ corresponds to a key range decided by the hash function $H$, the cardinality of $\mathsf{M}_i^j$ can also be estimated with the help of its histogram, which is denoted as $\widetilde{\mathsf{M}_i^j}$. Therefore, the value of $\omega_i$ can be estimated using Equation 5.2.

$$\omega_i = \sum_{j, W_j \text{ is active}} \widetilde{\mathsf{M}_i^j} * \omega_j \tag{5.2}$$

Observe that the estimated value of $\omega_i$ is calculated in an asynchronous manner, i.e., before each iteration starts. Therefore, the overhead to obtain $\omega_i$ is trivial compared with the overall evaluation process occurring in each iteration.

### 5.3.3 Theoretical Analysis

Finally, we provide a formal proof for the correctness of DWS strategy. While it is trivial to show the correctness for simple queries like *Same Generation*, the correctness for programs with `min` and `max` aggregates, that are viewed as extrema constraints in recursion, can be guaranteed by leveraging the PreM property [ZYD17]. The correctness is stated as Theorem 3.

**Theorem 3** *Let $P$ be a recursive Datalog program with $T$ as the corresponding ICO and $\gamma$ as the aggregate in recursion, we generate the parallel execution plan on $n$ workers using Global and DWS, respectively. If $\gamma$ is* PreM *to $T$, the parallel plan generated by DWS yields the same minimal fixpoint as that of Global, i.e. $\gamma(T^{\uparrow\omega}(\emptyset))$.*

For the proof of above Theorem, we begin from defining an operator $\trianglelefteq$ to denote the relationship between two sets. Given a recursive Datalog program $P$ with $T$ as the corresponding ICO, assume that $\gamma$ denotes $\texttt{min}\langle C\rangle$, or $\texttt{max}\langle C\rangle$, applied to the recursive table $S$ with group-by attributes $G$. If $\gamma$ is PreM to $T$ and $P$, then for two sets of tuples $S_1$ and $S_2$ with the same schema $S$, we define that $S_1 \trianglelefteq S_2$ if for every tuple $t_2 \in S_2$ there exists exactly one tuple $t_1 \in S_1$ s.t. $t_1[G] = t_2[G]$ and $\gamma(t_1[C], t_2[C]) = t_1[C]$. Note that in above definition, the commutative property does not hold for $\trianglelefteq$.

Let us next consider two intermediate results obtained during the semi-naive evaluation. If $\gamma$ stands for either `min` or `max`, and $T_\gamma(I)$ defines $\gamma(T(I))$, we have the following lemma:

**Lemma 1** *Given a recursive Datalog program $P$ with ICO $T$ where $\gamma$ is* PreM *to $T$, then for any two positive integers $x$, $y$ with $x \geq y$: $T^{\uparrow x}(\emptyset) \trianglelefteq T^{\uparrow y}(\emptyset)$.*

Proof: Due to the definition of semi-naive evaluation: $\forall s_1 \in T^{\uparrow x}(\emptyset)$ there exist a $s_2 \in T^{\uparrow y}(\emptyset)$, where $t_1[G] = t_2[G]$. Moreover, since $\gamma$ is PreM to $T$, it is obviously that $\gamma(t_1[C], t_2[C]) = t_1[C]$.

We can then further extend our discussion to the case described in Lemma 1 to parallel evaluation, which lead to Lemma 2.

**Lemma 2** *Given a recursive Datalog program $P$ with $T$ as the corresponding ICO and $\gamma$ is a* **min** *or* **max** *aggregate in recursion, we generate the parallel evaluation plan for $P$ on $n$ workers and $T_i$ is the ICO for worker $W_i$. Suppose $\gamma$ is* PreM *to $P$ and $T$, and for all $i \in [1, n]$, $\gamma$ is* PreM *to $T_i$. Let $I_g$ and $I_d$ are the interpretations of* Global *and* DWS *respectively. Then after $x(x > 0)$ rounds of iterations, $I_d \trianglelefteq I_g$ holds.*

Proof: During DWS based fixpoint computation, new tuples are produced by all workers $W_i$ in three ways: (i) From local computation of $W_i$; (ii) From join operation with a tuple fetched from another worker $W_j$ $(j \neq i)$; (iii) Form both (i) and (ii) together.

Then we can provide the proof by induction. For the base case, before the first coordination, each workers just perform one round of local iteration under Global. Meanwhile, a worker under DWS could perform more than one round of iterations. According to Lemma 1, $I_d \trianglelefteq I_g$ holds. Suppose for some $x \geq 1$ it is true that $I_d \trianglelefteq I_g$, under DWS each worker $W_i$ conducts the fixpoint computation on the tuples that generated from both $W_i$ in the previous iterations (case (i)) and the memory buffer of other workers (case (ii)) after $x$ rounds of iterations. In this process, for each $I_d$ and its corresponding $I_g$ we have $I_d \trianglelefteq I_g$. Therefore, $I_d \trianglelefteq I_g$ holds also for the $x+1$-th iteration. As a result, the lemma holds for all $x > 0$.

Finally, with the help of Lemma 2, we can reach the conclusion shown in Theorem 3, which demonstrate the correctness of DWS. The detailed proof of Theorem 3 is as following:

According to the description of PreM in [ZYD17], it is easy to observe that on other workers for all $i \in [1, n]$, $\gamma$ is also PreM to each $T_i$. Therefore, under the parallel evaluation plan generated by Global, it will yield the fixpoint $\gamma(T^{\uparrow\omega}(\emptyset))$. Meanwhile, under DWS any

tuple $t$ generated in any worker $W_i$ satisfies $t \in T^{\uparrow\omega}(\emptyset)$. That is, the interpretation under DWS $I_d$ is bounded, i.e. $I_d \subset T^{\uparrow\omega}(\emptyset)$. From Lemma 2, we can see that $I_d \trianglelefteq I_g$ holds. Since $\gamma(T^{\uparrow\omega}(\emptyset))$ is the least fixpoint under the $\gamma$ constraint, we also have $\gamma(T^{\uparrow\omega}(\emptyset)) \subset I_d$, as tuples in $\gamma(T^{\uparrow\omega}(\emptyset))$ should have the same values after aggregates are performed. Therefore we conclude from above discussion:

$$\gamma(T^{\uparrow\omega}(\emptyset)) \subset I_g \subset T^{\uparrow\omega}(\emptyset).$$

Furthermore, since $\gamma$ is PreM to each $T_i$, under DWS each worker $W_i$ also applies $\gamma$ in every iteration during the fixpoint computation. Thus we have: $I_g \subset \gamma(T^{\uparrow\omega}(\emptyset))$. By summing up above results and note that $I_g := \gamma(T^{\uparrow\omega}(\emptyset))$, we conclude that $I_g = I_d$. Therefore, the parallel plan generated by DWS yields the same minimal fixpoint $\gamma(T^{\uparrow\omega}(\emptyset))$ as that of Global.

For programs with `sum`, `count` and `average` in recursion, the correctness could be ensured with the PCC property in Chapter 4 in a similar way.

## 5.4 Evaluation

### 5.4.1 Experiment Setup

#### 5.4.1.1 Benchmark Programs and Datasets

Table 5.1: Graph and Network Datasets

| Name | # Vertices | # Edges | Size |
|---|---|---|---|
| LiveJournal | 4,847,572 | 68,993,773 | 527 MB |
| Orkut | 3,072,441 | 117,185,083 | 895 MB |
| Arabic | 22,744,080 | 639,999,458 | 4.8 GB |
| Twitter | 41,652,231 | 1,468,365,182 | 11 GB |

To evaluate our proposed DCDatalog engine, we conduct experiments using five Datalog programs which were widely used in previous studies. The first two are *Same Generation*

*(SG)* and *Delivery* [bom] (Query 7-8) that were popular examples for deductive database; the other three express the following popular graph algorithms (Query 6, 9-10): *Connected Component*, *PageRank* and *Single Source Shortest Path*.

*Query 7* - Same Generation (SG)

$$\begin{aligned}
\mathsf{r}_{3,1}: \quad sg(X,Y) &\leftarrow arc(P,X), arc(P,Y), X! = Y.\\
\mathsf{r}_{3,2}: \quad sg(X,Y) &\leftarrow arc(A,X), sg(A,B), arc(B,Y).
\end{aligned}$$

*Query 8* - BoM – Delivery

$$\begin{aligned}
\mathsf{r}_{6,1}: \quad delivery(P, max\langle D\rangle) &\leftarrow basic(P,D).\\
\mathsf{r}_{6,2}: \quad delivery(P, max\langle D\rangle) &\leftarrow assbl(P,S),\\
&\quad\; delivery(S,D).\\
\mathsf{r}_{6,3}: \quad results(P, max\langle D\rangle) &\leftarrow delivery(part, D).
\end{aligned}$$

*Query 9* - PageRank (PR)

$$\begin{aligned}
\mathsf{r}_{4,1}: \quad rank(X, sum\langle(X,I)\rangle) &\leftarrow matrix(X, _, _),\\
&\quad\; I = (1-\alpha)/VNUM.\\
\mathsf{r}_{4,2}: \quad rank(X, sum\langle(Y,K)\rangle) &\leftarrow rank(Y,C),\\
&\quad\; matrix(Y,X,D),\\
&\quad\; K = \alpha * (C/D).\\
\mathsf{r}_{4,3}: \quad results(X,V) &\leftarrow rank(X,V).
\end{aligned}$$

*Query 10* - Single Source Shortest Path (SSSP)

$$\begin{aligned}
\mathsf{r}_{5,1}: \quad sp(To, min\langle C\rangle) &\leftarrow To = start, C = 0.\\
\mathsf{r}_{5,2}: \quad sp(To2, min\langle C\rangle) &\leftarrow sp(To1, C1),\\
&\quad\; warc(To1, To2, C2),\\
&\quad\; C = C1 + C2.\\
\mathsf{r}_{5,3}: \quad results(To, min\langle C\rangle) &\leftarrow sp(To, C).
\end{aligned}$$

For the three graph queries, we evaluate them on four real world datasets Livejournal, Orkut, Arabic and Twitter, whose detailed statistics are shown in Table 5.1. For the first two queries, we evaluate on synthetic datasets used in previous studies [SYI16, FZZ19, GWM19]: Tree-11 is a tree of height 11, and the degree of a non-leaf vertex is a random number between 2 and 6. G-10K is a 10,000-vertex random graph [4] generated by randomly connecting vertices so that each pair is connected with probability 0.001. The Rmat-$n$ graphs are generated by the RMAT graph generator, which has $n$ vertices and $10 \times n$ directed edges. The N-$n$ are trees with $n$ vertices, which are generated in different levels following [GWM19]: each tree node has randomly 5 to 10 children, and each child has a 20% to 60% chance of becoming a leaf.

#### 5.4.1.2 Baseline Systems

We used the following Datalog engines designed for shared-memory multicore architectures as the baseline for our work: SociaLite [SGL13], DeALS-MC [YSZ17], Souffle [SJS16] equipped with the concurrent index [JSZ19] and RecStep [FZZ19]. We also compared with the BigDatalog [SYI16] engine which works under shared-nothing architecture to further demonstrate the significance of these results in the wider context of parallel system.

For above baseline systems, we obtained the source code of DeALS-MC and SociaLite from original authors. The codes of Souffle[5], BigDatalog[6], and RecStep[7] are public available.

The rationale for focusing on the those Datalog systems, and excluding a few others from our comparisons, is based on the following considerations. It has been shown in [YSZ17] that the single-node based DeALS [SYZ15] and LogicBlox [ACG15] cannot outperform DeALS-

[4]http://www.cse.psu.edu/ kxm85/software/GTgraph

[5]https://souffle-lang.github.io/

[6]We fix some bugs in this version to support *PageRank*: https://github.com/ashkapsky/BigDatalog

[7]https://github.com/Hacker0912/RecStep

MC. Meanwhile, BigDatalog has significantly better performance than Myria [WBH15] and Distributed SociaLite [SPS13]. Furthermore, we have not considered specialized non-Datalog systems such as special-purpose graph systems.

We use the end-to-end query execution time as the metric for evaluation. Since in this paper we focus on in-memory computation, we exclude the time of loading data for all the systems, which is in fact rather trivial for DCDatalog. We run all the experiments 5 times and report the average results. If a system cannot finish within 10 hours under a particular setting, we will regard it as timeout.

#### 5.4.1.3 Environment

We implement the DCDatalog engine with C++. We run the experiments of all the systems except BigDatalog on a server with four AMD Opteron 6376 CPUs (8 physical cores per CPU, 2 hyper-thread per core), 256GB memory (configured into eight NUMA regions) and 1 TB hard disk. The operating system is Ubuntu Linux 14.04 LTS and the compiler is GCC 9.0 with O3 flag. For BigDatalog, we conduct the experiments on a cluster with 16 nodes. 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. Each worker node is allocated 30 GB RAM and 8 CPU cores (120 total cores) for execution.

Table 5.2: Comparison with State-of-the-art Systems (seconds): OOM means out of memory; NS means the system does not support the corresponding query; TO means timeout

| Query | Dataset | DCDatalog | SociaLite | DeALS-MC | Souffle | RecStep | BigDatalog |
|---|---|---|---|---|---|---|---|
| *SG* | Tree-11 | 40.77 | 30687.42 | 71.99 | 1438.98 | OOM | 53.40 |
| | G-10K | 16.04 | 4762.25 | 76.18 | 194.09 | 458.41 | 95.32 |
| | Rmat-10K | 12.18 | 5013.76 | 80.11 | 143.46 | 512.48 | 108.17 |
| | Rmat-20K | 54.90 | 21048.49 | 299.16 | 664.65 | 2378.16 | 577.65 |
| | Rmat-40K | 237.07 | TO | 1358.42 | 2879.03 | OOM | OOM |
| *Delivery* | N-40M | 3.32 | 233.71 | NS | 88.06 | 40.26 | 12.57 |
| | N-80M | 5.15 | 854.73 | NS | 167.67 | 71.71 | 15.69 |
| | N-160M | 11.28 | 2332.05 | NS | 369.81 | 154.13 | 18.35 |
| | N-300M | 18.78 | 8170.65 | NS | 729.52 | 334.43 | 28.17 |
| *CC* | Livejournal | 8.49 | 31.70 | 319.88 | OOM | 55.12 | 27.98 |
| | Orkut | 11.09 | 40.91 | 379.30 | OOM | 49.41 | 31.78 |
| | Arabic | 50.58 | 184.55 | OOM | OOM | 495.54 | 213.59 |
| | Twitter | 77.84 | TO | OOM | OOM | 637.51 | 307.69 |
| *SSSP* | Livejournal | 11.93 | 42.36 | 791.83 | OOM | 212.50 | 53.80 |
| | Orkut | 8.66 | 36.84 | 361.71 | OOM | 88.01 | 39.47 |
| | Arabic | 9.90 | 61.69 | OOM | OOM | 113.96 | 276.55 |
| | Twitter | 24.01 | TO | OOM | OOM | 178.24 | 260.71 |
| *PageRank* | Livejournal | 113.42 | 12339.52 | NS | OOM | NS | 135.87 |
| | Orkut | 45.71 | 4770.41 | NS | OOM | NS | 88.77 |
| | Arabic | 203.94 | TO | NS | OOM | NS | 680.00 |
| | Twitter | 2020.85 | TO | NS | OOM | NS | 2356.57 |

### 5.4.2 End-to-end Query Time Comparison

We first report results of comparing with existing Datalog engines as shown in Table 5.2. For the two recursive queries *SG* and *Delivery*, we can find that DCDatalog achieves 3 to 100 times performance gain over the baselines. For example, for the *SG* query on G-10K dataset, the times for Souffle, RecStep, DeALS-MC and SociaLite are 194.09, 458.41, 76.18 and 4762.25 seconds, respectively. Meanwhile, DCDatalog takes only 16.04 seconds. The superior performance of DCDatalog comes from the comprehensive optimizations made in all components of the system. A separate issue is that some of the language constructs of DCDatalog are not supported in other systems. For instance, Souffle does not allow aggregates in recursion, and thus it must use a stratified query has very poor performance to express the *Delivery* query. Compared with DeALS-MC, we adopt DWS approach for coordination between different workers and thus can save the time for idle waiting. The performance of SociaLite queries underscores that the system was optimized for social network applications rather than general-purpose Datalog queries. For RecStep, the source code released by the author does not include the claimed PBME optimizations in [FZZ19]. Therefore, we just report the results we obtained from their currently released version, which are likely to be worse than those reported in the original paper.

We further look at the results on the three graph algorithms *CC*, *SSSP* and *PageRank*. The trends are similar to those observed in the two recursive queries we just discussed. Many baseline systems, such as DeALS-MC and RecStep, cannot support *PageRank* because they fail to support expressing the `sum` aggregate in recursion. Souffle runs out of memory on all graph queries because the equivalent stratified queries involve too many intermediate results. Compared with other baseline systems, DCDatalog has both great expressive power and performance because it relaxes the constraint on lock-free programs and uses a light-weight scheme to deal with the race condition that happens in evaluating programs with complicated aggregates in recursion. For instance, for the *SSSP* query on ORKUT dataset, the time for RecStep, DeALS-MC and SociaLite is 88.01, 361.71 and 36.84 seconds, respectively; while

that for DCDatalog is 8.66 seconds. Even compared with the BigDatalog system which runs in the shared-nothing environment, DCDatalog still has better performance on all queries in most settings.

### 5.4.3 Micro-Benchmarking Results

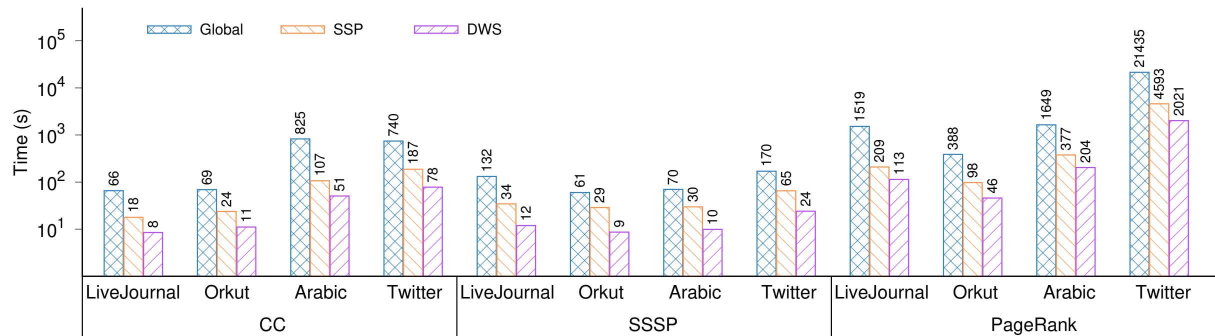



Figure 5.4: Effect of Different Coordination Strategies

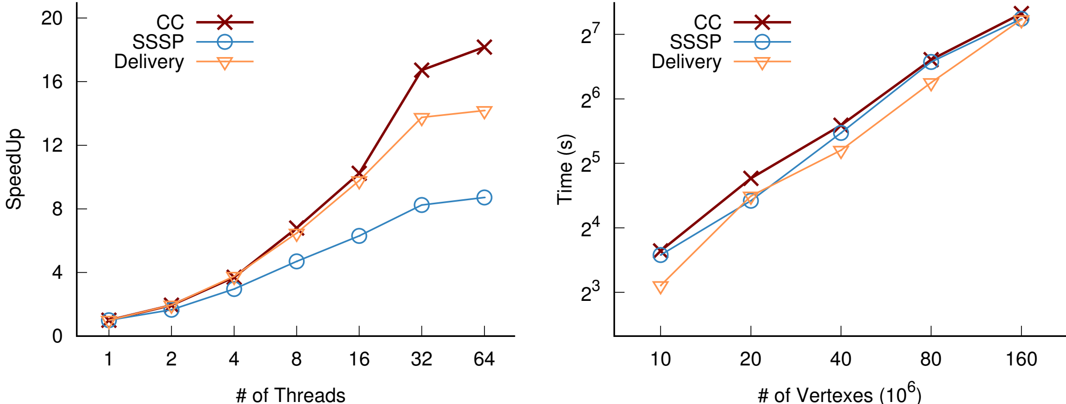
Next we dive into the details of how the various optimization techniques improve query performance. We aim at assessing the effectiveness of the parallel coordination strategy DWS (Section 5.3.2).

To test the effect of different coordination strategies, we consider the following three methods: Global is the method that requires a coordination after each global iteration; SSP simply extends the techniques proposed in [DZ19] designed for shared-nothing environment, which allows fast workers to proceed at most $s$ iterations; DWS is our proposed dynamic coordination strategy. In this experiment, we set $s$ as 5 empirically, which has the best performance under most settings. The results of different coordination strategies are shown in Figure 5.4. We see that DWS achieves the best performance under all settings. For example, for the *SSSP* query on LIVEJOURNAL dataset, the time for Global, SSP and DWS is 131.68, 34.45 and 11.93 seconds, respectively. SSP performs worse than DWS because it relies on a predefined threshold $s$ to avoid the idle waiting of faster workers, which fails

to reflect the characteristics of different iterations during the evaluation. Global has the worst performance as it suffers from the idle waiting involved in the parallel evaluation. Note that Global uses the same coordination strategy with DeALS-MC but is equipped with better implementation techniques by us. Therefore, its general performance is better than DeALS-MC due to the benefits of our designs.

### 5.4.4 Scalability

Finally we evaluate the scalability of DCDatalog.




(a) Varying # Threads (b) Varying Data Size

Figure 5.5: Scalability: Datalog on Multicore Machines

We first conduct experiments of scaling up the number of workers (threads). We evaluate the *CC*, *SSSP* and *Delivery* programs on dataset LIVEJOURNAL, ARABIC, and N-300M, respectively. We vary the number of threads from 1 to 64. The results are shown in Figure 5.5(a). We observe that DCDatalog scales well using up to 32 threads. After that, the speedup tends to be stable because the number of physical cores on the machine is 32 and they are fully utilized. *SSSP* query achieves smaller speedup since its evaluation starts from one vertex and is not fully paralleled in a long period after beginning.

Then we increase the size of dataset and observe how our system scales over large volume of data. We use the synthetic dataset RMAT-$n$ by varying the number of vertices in the

generated graph from 10M to 160M. Figure 5.5(b) shows the results of *CC*, *SSSP* and *Delivery* programs. We observe that the time increases proportionally to the size of datasets. For example, the execution time of *CC* is 12.48, 27.26, 48.23, 97.55, 160.19 seconds on the synthetic graph with 10M, 20M, 40M, 80M, 160M vertices, respectively. It shows the potential ability of DCDatalog for dealing with ever larger datasets on modern multicore machines.

## 5.5 Related Work

### 5.5.1 Datalog Language and Evaluation

Supporting aggregates in recursive Datalog programs is an old and difficult problem which has been the topic of much previous research work. Earlier studies tried to reach this goal by providing formal semantics for recursive Datalog programs with unstratified aggregates [GGZ91, MPR90, FGG02]. In particular, Ross et al. [RS92] used semantics based on specialized lattices to express the four aggregates, while Ganguly et al. [GGZ95] sought to optimize programs with extrema. Mazuran et al. [MSZ13] proposed the monotonic aggregates and prove that they can be used freely in recursion. More recently, Zaniolo et al. [ZYD17] introduced the Pre-mappability(PreM) property under which programs using extrema in recursion are equivalent to those aggregate-stratified ones. It also enables the RaSQL language [GWM19, WXG20], a recursive SQL that supports aggregates in recursion.

There is a long stream of studies about supporting parallel evaluation of recursive Datalog programs. Wolfson et al. [WS88] identified the decomposable programs which can be evaluated in parallel without communication and duplicated computation. The parallel SN evaluation fixpoint was proposed in [GST90] for message passing. Seib et al. [SL91] provided the Generalized Pivoting to divide the workload in Datalog program for parallel execution and Ganguly et al. [GST92] proposed the substitution partitioned parallelization scheme. Shaw et al. [SKH12] and Afrati et al. [ABC11, AU12] studied how to support Datalog eval-

uation under MapReduce framework. Motik et al. [MNP14] focused on the specific problem of RDF data. All these studies focused more on theoretical results rather than providing concrete system implementation.

### 5.5.2 Datalog Systems and Applications

Many efforts have been paid to design and implement an efficient engine for Datalog evaluation. LogicBlox [ACG15] designed the Datalog engine according to the similar idea of relational DBMS. DeALS [SYZ15] implemented the idea of monotonic aggregation to efficiently support aggregates in recursion. To deal with large-scale analytical queries, another line of studies focus on developing distributed Datalog engines. Distributed SociaLite [SPS13] extended its single node version [SGL13] to shared-nothing environment with message passing to communicate. Myria [WBH15] proposed an asynchronous approach for Datalog evaluation. BigDatalog [SYI16] developed the Datalog engine on top of Apache Spark. There are also some special purposed systems that use Datalog-like interfaces due to its succinct program structure and superior expressive power to support a wide spectrum of applications, such as knowledge reasoning [BSG18], graph analysis [ATO16], program analysis [WAC05] and data center management [ZAC19].

For the systems in shared-memory architectures, DeALS-MC [YSZ17] implements and optimizes the idea of *substitution partitioned parallelization* for lock-free programs. However, it has certain limitations of performance due to its coordination strategy, which has been detailed in Section 5.2.1. Souffle [SJS16, JSZ19] is a Datalog engine designed with concurrent B-Tree indexes. It cannot support aggregates in recursion and thus fail to express many advanced analytical queries. RecStep [FZZ19] is a parallel Datalog engine implemented on top a parallel relational database system named QuickStep [PDZ18], which is responsible to support the parallel execution of the analyzed Datalog programs. The RecStep engine itself did not propose techniques for improving the parallel evaluation as our work did.

### 5.5.3 Parallel Query Evaluation

In the past years, many distributed big data platforms have been developed to cope with the ever-increasing volume of data collections. For distributed big data platforms, a critical bottleneck is the synchronization mechanism over all workers. The Bulk Synchronous Parallel (BSP) model is the most popular one for distributed computation. Under BSP, iterative computation is separated into super steps, and messages from one super step are only accessible in the consequent one. It has been adopted by both general purpose systems like Apache Spark [ZCD12] and graph processing systems, such as Pregel [MAB10] and GraphX [GXD14]. To alleviate the overhead of synchronization in BSP, some other systems adopted the Asynchronous Processing (AP) model, such as GraphLab [LGK12] and Giraph++ [TBC13]. Some follow-up studies [CCH14, XCG15, HD15, FLL18] targeted at making a trade-off between AP and BSP to propose new synchronization techniques, which can reduce both the cost of global synchronization and communication overheads. Above strategies are all designed for applications of graph analysis or machine learning in a shared-nothing environment, which was not the focus of our study. It is an interesting direction to investigate how to extend them to our problem in the future work.

## 5.6 Conclusion of Chapter

In this paper, we introduce DCDatalog, a parallel Datalog engine on shared-memory multicore architectures. DCDatalog is equipped with a light-weight scheme to resolve race conditions in parallel execution, thus enabling more efficient evaluation for a broad range of Datalog applications. We propose a novel dynamic coordination strategy to overcome the limitations of existing approaches for parallel Datalog evaluation. The proposed strategy significantly reduces the idle waiting time and brings additional benefits to recursive queries. Experimental results on several real world datasets demonstrate the superior efficiency and scalability of DCDatalog compared with existing Datalog engines.

# CHAPTER 6

# Conclusion and Future Work

The recent new findings in Datalog research enable the use of aggregates in recursion, and this has brought a revival of interest in Datalog for expressing more powerful data-intensive applications. In this dissertation, we have addressed several research problems from language semantics to applications and performance driven by such recently introduced notions in *Datalog*.

To begin with, we propose a declarative framework to support machine learning applications on Apache Spark. The proposed framework (i) expresses popular ML applications with succinct *Datalog* programs and user friendly DataFrame APIs; (ii) provides effective compilation and planning techniques to support complex recursions in ML applications; (iii) devises efficient and automatic optimizations to improve the overall performance. We perform an extensive set of experimental study on both synthetic and real world datasets. The results demonstrate the superior performance and scalability of our framework over several large-scale datasets.

In addition, we present the Pre-Countable Cardinality(PCC) Property, which provides formal semantics for a complete set of aggregates in recursion. We find that set aggregates can be used inside the recursion when the cardinality of involved relations can be pre-computed before the recursive computation begins, and simply passed to the fixpoint computation that follows. We then use several examples to show how this property can benefit a wide spectrum of data analysis applications, especially complicated machine learning and data mining algorithms.

Last but not least, we design and implement a prototype system for scaling up *Datalog* evaluation on shared-memory multi-core machine. We propose a novel coordination strategy to improve the parallelism of semi-naive evaluation by avoiding unnecessary idle waiting and providing finer granularity concurrency management. The experimental results show that our system outperforms other coordination strategies by an obvious margin.

The research findings in this dissertation also open up new opportunities in several research directions. First of all, we believe the usage of completed aggregates in recursive rules made possible by the PCC property can lead to further extensions in a variety of applications, such as natural language understanding, data integration and data mining. Besides, some of my previous research works lies in the applications of many data types, such as text [WLL20, ZWW20, TZW19, WLZ19, WLL19], spatial [WZW19, YZZ19], and streaming [DWG19, ASW19, GWZ16]. Nowadays, machine learning techniques have been widely adopted in such domains, which brings new challenges in expressiveness and performance. Thus, it is worthy to investigate how to take advantage of the expressive power of *Datalog* to express queries over such a broad scope of applications and provide efficient and scalable implementations for these powerful tools.

## REFERENCES


[ABC11] Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. "Map-reduce extensions and recursive queries." In *EDBT*, pp. 1–8, 2011.

[ABC16] Martín Abadi, Paul Barham, Jianmin Chen, and et al. "TensorFlow: A System for Large-Scale Machine Learning." In *OSDI*, pp. 265–283, 2016.

[ACG15] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. "Design and Implementation of the LogicBlox System." In *SIGMOD*, pp. 1371–1382, 2015.

[AGN13] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony I. T. Rowstron. "Scale-up vs scale-out for Hadoop: time to rethink?" In *SOCC*, pp. 20:1–20:13, 2013.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

[ASS17] Michael J. Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capota, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L. Willke. "Bridging the Gap between HPC and Big Data frameworks." *PVLDB*, **10**(8):901–912, 2017.

[ASW19] Xiang Ao, Haoran Shi, Jin Wang, Luo Zuo, Hongwei Li, and Qing He. "Large-Scale Frequent Episode Mining from Complex Event Sequences with Hierarchies." *ACM TIST*, **10**(4):36:1–36:26, 2019.

[ATO16] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. "EmptyHeaded: A Relational Engine for Graph Processing." In *SIGMOD*, pp. 431–446, 2016.

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

[BBC12a] Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. "Declarative Systems for Large-Scale Machine Learning." *IEEE Data Eng. Bull.*, **35**(2):24–32, 2012.

[BBC12b] 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.

[BDE16] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. “SystemML: Declarative Machine Learning on Spark.” *PVLDB*, **9**(13):1425–1436, 2016.

[bom] “Recursion Example: Bill Of Materials.” `https://www.ibm.com/support/knowledgecenter/en/SS6NHC/com.ibm.swg.im.dashdb.sql.ref.doc/doc/r0059242.html`.

[BSG18] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. “The Vadalog System: Datalog-based Reasoning for Knowledge Graphs.” *PVLDB*, **11**(9):975–987, 2018.

[CCH14] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. “Exploiting Bounded Staleness to Speed Up Big Data Analytics.” In *USENIX ATC*, pp. 37–48, 2014.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. “An Overview of Data Warehousing and OLAP Technology.” *SIGMOD Record*, **26**(1):65–74, 1997.

[CDI18] Tyson Condie, Ariyam Das, Matteo Interlandi, Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. “Scaling-up reasoning and advanced analytics on BigData.” *TPLP*, **18**(5-6):806–845, 2018.

[CEF17] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. “State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing.” *PVLDB*, **10**(12):1718–1729, 2017.

[CKN17] Lingjiao Chen, Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. “Towards Linear Algebra over Normalized Data.” *PVLDB*, **10**(11):1214–1225, 2017.

[CLL15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems.” *CoRR*, **abs/1512.01274**, 2015.

[CVP13] Zhuhua Cai, Zografoula Vagena, Luis Leopoldo Perez, Subramanian Arumugam, Peter J. Haas, and Christopher M. Jermaine. “Simulation of database-valued markov chains using SimSQL.” In *SIGMOD*, pp. 637–648, 2013.

[Dar20] Adnan Darwiche. “Three Modern Roles for Logic in AI.” *PODS*, 2020.

[DLW19] Ariyam Das, Youfu Li, Jin Wang, Mingda Li, and Carlo Zaniolo. "BigData Applications from Graph Analytics to Machine Learning by Aggregates in Recursion." In *ICLP*, pp. 273–279, 2019.

[DWG19] Ariyam Das, Jin Wang, Sahil M. Gandhi, Jae Lee, Wei Wang, and Carlo Zaniolo. "Learn Smart with Less: Building Better Online Decision Trees with Fewer Training Examples." In *IJCAI*, pp. 2209–2215, 2019.

[DZ19] Ariyam Das and Carlo Zaniolo. "A Case for Stale Synchronous Distributed Model for Declarative Recursive Computation." *TPLP*, **19**(5-6):1056–1072, 2019.

[EBH16] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. "Compressed Linear Algebra for Large-Scale Machine Learning." *PVLDB*, **9**(12):960–971, 2016.

[ELB17] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. "SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning." In *CIDR*, 2017.

[FGG02] Filippo Furfaro, Sergio Greco, Sumit Ganguly, and Carlo Zaniolo. "Pushing extrema aggregates to optimize logic queries." *Inf. Syst.*, **27**(5):321–343, 2002.

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

[FLL18] Wenfei Fan, Ping Lu, Xiaojian Luo, Jingbo Xu, Qiang Yin, Wenyuan Yu, and Ruiqi Xu. "Adaptive Asynchronous Parallelization of Graph Algorithms." In *SIGMOD*, pp. 1141–1156, 2018.

[FZZ19] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. "Scaling-Up In-Memory Datalog Processing: Observations and Techniques." *PVLDB*, **12**(6):695–708, 2019.

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

[GGZ95] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. "Extrema Predicates in Deductive Databases." *J. Comput. Syst. Sci.*, **51**(2):244–259, 1995.

[GLP17] Zekai J. Gao, Shangyu Luo, Luis Leopoldo Perez, and Chris Jermaine. "The BUDS Language for Distributed Bayesian Machine Learning." In *SIGMOD*, pp. 961–976, 2017.

[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.

[GWM19] Jiaqi Gu, Yugo Watanabe, William Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. "RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-aggregate-SQL on Spark." In *SIGMOD*, pp. 467–484, 2019.

[GWZ16] Jiaqi Gu, Jin Wang, and Carlo Zaniolo. "Ranking support for matched patterns over complex event streams: The CEPR system." In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pp. 1354–1357, 2016.

[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.

[HCC13] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server." In *NIPS*, pp. 1223–1231, 2013.

[HD15] Minyang Han and Khuzaima Daudjee. "Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems." *PVLDB*, **8**(9):950–961, 2015.

[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.

[JLY19] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. "Declarative Recursive Computation on an RDBMS." *PVLDB*, **12**(7):822–835, 2019.

[JSZ19] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. "A specialized B-tree for concurrent datalog evaluation." In *PPoPP*, pp. 327–339, 2019.

[JZC16] Yu-Chin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. "Field-aware Factorization Machines for CTR Prediction." In *RecSys*, pp. 43–50, 2016.

[KQT17] Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Saravanan Thirumuruganathan, Sanjay Chawla, and Divy Agrawal. "A Cost-based Optimizer for Gradient Descent Optimization." In *SIGMOD*, pp. 977–992, 2017.

[KTD13] Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. "MLbase: A Distributed Machine-learning System." In *CIDR*, 2013.

[LAP14] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. "Scaling Distributed Machine Learning with the Parameter Server." In *OSDI*, pp. 583–598, 2014.

[LCC17] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. "MLog: Towards Declarative In-Database Machine Learning." *PVLDB*, **10**(12):1933–1936, 2017.

[LCG06] 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: language, execution and optimization." In *SIGMOD*, pp. 97–108, 2006.

[LCH05] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. "Implementing declarative overlays." In *SOSP*, pp. 75–90, 2005.

[LGG17] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, and Christopher M. Jermaine. "Scalable Linear Algebra on a Relational Database System." In *ICDE*, pp. 523–534, 2017.

[LGK12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning in the Cloud." *PVLDB*, **5**(8):716–727, 2012.

[lib] "LIBSVM Data ." `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

[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.

[mah] "Apache Mahout ." `https://mahout.apache.org/`.

[MBY16] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, and et al. "MLlib: Machine Learning in Apache Spark." *Journal of Machine Learning Research*, **17**:34:1–34:7, 2016.

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

[MMI13] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. "Differential Dataflow." In *CIDR*, 2013.

[MNP14] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. "Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems." In *AAAI*, pp. 129–137, 2014.

[MNW18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. "Ray: A Distributed Framework for Emerging AI Applications." In *OSDI*, pp. 561–577, 2018.

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

[MSS09] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. "Identifying suspicious URLs: an application of large-scale online learning." In *ICML*, pp. 681–688, 2009.

[MSZ13] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. "Extending the power of datalog recursion." *VLDB J.*, **22**(4):471–493, 2013.

[MVP18] Nantia Makrynioti, Nikolaos Vasiloglou, Emir Pasalic, and Vasilis Vassalos. "Modelling Machine Learning Algorithms on Relational Data with Datalog." In *DEEM@SIGMOD*, pp. 5:1–5:4, 2018.

[PDZ18] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. "Quickstep: A Data Platform Based on the Scaling-Up Approach." *PVLDB*, **11**(6):663–676, 2018.

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

[SAD10] Michael Stonebraker, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. "MapReduce and parallel DBMSs: friends or foes?" *Commun. ACM*, **53**(1):64–71, 2010.

[SDC19] Benoit Steiner, Zachary DeVito, Soumith Chintala, and et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In *NeurIPS*, 2019.

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

[SJS16] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. "On fast large-scale program analysis in Datalog." In *CC*, pp. 196–206, 2016.

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

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

[SOC16] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. "Learning Linear Regression Models over Factorized Joins." In *SIGMOD*, pp. 3–18, 2016.

[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.

[STS19] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. "Presto: SQL on Everything." In *ICDE*, pp. 1802–1813, 2019.

[SV17] Umar Syed and Sergei Vassilvitskii. "SQML: large-scale in-database machine learning with pure SQL." In *SoCC*, p. 659, 2017.

[SVK17] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. "KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics." In *ICDE*, pp. 535–546, 2017.

[SYI16] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. "Big Data Analytics with Datalog Queries on Spark." In *SIGMOD*, pp. 1135–1149, 2016.

[SYZ15] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. "Optimizing recursive queries with monotonic aggregates in DeALS." In *ICDE*, pp. 867–878, 2015.

[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.

[TK18] Anthony Thomas and Arun Kumar. "A Comparative Evaluation of Systems for Scalable Linear Algebra-based Analytics." *PVLDB*, **11**(13):2168–2182, 2018.

[TSJ10] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. "Hive - a petabyte scale data warehouse using Hadoop." In *ICDE*, pp. 996–1005, 2010.

[TZW19] Bing Tian, Yong Zhang, Jin Wang, and Chunxiao Xing. "Hierarchical Inter-Attention Network for Document Classification with Multi-Task Learning." In *IJCAI*, pp. 3569–3575, 2019.

[WAC05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. "Using Datalog with Binary Decision Diagrams for Program Analysis." In *APLAS*, pp. 97–118, 2005.

[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.

[WCP06] Steve Webb, James Caverlee, and Calton Pu. "Introducing the Webb Spam Corpus: Using Email Spam to Identify Web Spam Automatically." In *CEAS*, 2006.

[WLL19] Jin Wang, Chunbin Lin, Mingda Li, and Carlo Zaniolo. "An Efficient Sliding Window Approach for Approximate Entity Extraction with Synonyms." In *EDBT*, pp. 109–120, 2019.

[WLL20] Jin Wang, Chunbin Lin, Mingda Li, and Carlo Zaniolo. "Boosting approximate dictionary-based entity extraction with synonyms." *Inf. Sci.*, **530**:1–21, 2020.

[WLZ19] Jin Wang, Chunbin Lin, and Carlo Zaniolo. "MF-Join: Efficient Fuzzy String Similarity Join with Multi-level Filtering." In *ICDE*, pp. 386–397, 2019.

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

[WXG20] Jin Wang, Guorui Xiao, Jiaqi Gu, Jiacheng Wu, and Carlo Zaniolo. "RASQL: A Powerful Language and its System for Big Data Applications." In *SIGMOD*, pp. 2673–2676, 2020.

[WZW19] Jiacheng Wu, Yong Zhang, Jin Wang, Chunbin Lin, Yingjia Fu, and Chunxiao Xing. "Scalable Metric Similarity Join Using MapReduce." In *ICDE*, pp. 1662–1665, 2019.

[XCG15] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. "SYNC or ASYNC: time to fuse for distributed graph-parallel computation." In *PPoPP*, pp. 194–204, 2015.

[XHD15] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. "Petuum: A New Platform for Distributed Machine Learning on Big Data." In *ACM SIGKDD*, pp. 1335–1344, 2015.

[XMM18] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. "Helix: Holistic Optimization for Accelerating Iterative Machine Learning." *PVLDB*, **12**(4):446–460, 2018.

[YBT17] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. "Big Graph Analytics Platforms." *Found. Trends Databases*, **7**(1-2):1–195, 2017.

[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.

[YSZ17] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. "Scaling up the performance of more powerful Datalog systems on multicore machines." *VLDB J.*, **26**(2):229–248, 2017.

[YZZ19] Junye Yang, Yong Zhang, Xiaofang Zhou, Jin Wang, Huiqi Hu, and Chunxiao Xing. "A Hierarchical Framework for Top-k Location-Aware Error-Tolerant Keyword Search." In *ICDE*, pp. 986–997, 2019.

[ZAC19] Qizhen Zhang, Akash Acharya, Hongzhi Chen, Simran Arora, Ang Chen, Vincent Liu, and Boon Thau Loo. "Optimizing Declarative Graph Queries at Large Scale." In *SIGMOD*, pp. 1411–1428, 2019.

[ZBW12] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. "SCOPE: parallel databases meet MapReduce." *VLDB J.*, **21**(5):611–636, 2012.

[ZCD12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." In *NSDI*, pp. 15–28, 2012.

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

[ZWW20] Yong Zhang, Jiacheng Wu, Jin Wang, and Chunxiao Xing. "A Transformation-Based Framework for KNN Set Similarity Search." *IEEE Trans. Knowl. Data Eng.*, **32**(3):409–423, 2020.

[ZYD17] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. "Fixpoint semantics and optimization of recursive Datalog programs with aggregates." *TPLP*, **17**(5-6):1048–1065, 2017.

[ZYI18] Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, Alexander Shkapsky, and Tyson Condie. "Declarative BigData Algorithms via Aggregates and Relational Database Dependencies." In *AWM*, 2018.