UNIVERSITY OF CALIFORNIA
Los Angeles

Query Language Extensions for Advanced Analytics on Big Data and their Efficient Implementation

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

by

Jiaqi Gu

2019

© Copyright by

Jiaqi Gu

2019

ABSTRACT OF THE DISSERTATION

Query Language Extensions for Advanced Analytics on Big Data
and their Efficient Implementation

by

Jiaqi Gu
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2019
Professor Carlo Zaniolo, Chair

Advanced analytics and other Big Data applications call for query languages that can express the complex logic of advanced analytics, and are also amenable to efficient implementations providing high throughput and low latency. Existing systems such as Hadoop or Spark can now handle large amounts of data via MapReduce enabled parallelism, but they lack simple query languages that can express declaratively applications such as common graph and data mining algorithms, and the search for complex patterns in massive data sets. Fortunately, recent advances in recursive query languages and automata theory have paved way for extending widely used declarative query languages, such as SQL, to address these problems. Thus, in this dissertation, we propose two significant new extensions to the current SQL standards and demonstrate their efficient implementations. We first propose the Recursive-aggregate-SQL language, RaSQL. RaSQL queries assure a declarative formal fixpoint semantics that is guaranteed by the PreM property, while amenable to efficient recursive query evaluation techniques based on the Semi-Naïve optimization for the fixpoint computation. The RaSQL is implemented on top of Apache Spark, achieving superior scalability and performance compared to the state-of-art systems such as Apache Giraph, GraphX and Myria. Then, we propose a new Weighted Search Pattern language, WSP, which extends the SQL-TS language. WSP is able to provide semantic rankings of the query results, and its implementation and optimization are guided by the theory of weighted automata.

The dissertation of Jiaqi Gu is approved.

Yingnian Wu

Todd D. Millstein

Junghoo Cho

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

2019

To my wife and my parents

TABLE OF CONTENTS

1 Introduction **1**

2 Background **5**

2.1 Datalog 5

2.2 Recursive SQL 8

2.3 Apache Spark and Spark SQL 9

3 RaSQL: Greater Power and Performance with Recursive-aggregate-SQL **11**

3.1 Introduction 11

3.2 The RaSQL Language 14

3.3 The PreM property 18

3.3.1 Programming with Aggregates in Recursion 20

3.3.2 Testing PreM 22

3.4 RaSQL Examples 24

3.5 Compiler Implementation 31

3.5.1 Recursive Clique Analysis 32

3.5.2 Physical Plan Generation 33

3.6 Fixpoint Operator Execution 34

3.6.1 Distributed Semi-Naive Evaluation 37

3.6.2 Aggregates In Recursion 40

3.6.3 Join Implementation 41

3.7 Performance Optimization 42

3.7.1 Stage Combination 43

3.7.2 Decomposed Plan Optimization 45

3.7.3 Code Generation 47

3.8 Experiments 48

3.8.1 Graph Data Analytics 50

3.8.2 Complex Data Analytics 54

3.8.3 Scaling Experiments . . . 55
3.8.4 Performance Comparison with Single-Node Implementation . . . 55
3.9 Related Work . . . 57

4 A Weighted Search Pattern Language and its Efficient Implementation 59
4.1 Introduction . . . 60
4.2 WSP Query Language . . . 62
4.3 Formal Evaluation Model . . . 70
4.3.1 Weighted Automaton . . . 70
4.3.2 Compilation using Weighted Automaton . . . 72
4.3.3 Evaluation Semantics . . . 74
4.4 WSP Optimization . . . 75
4.4.1 Compile-time Optimization . . . 75
4.4.2 Run-time Optimization . . . 76
4.5 Experiments . . . 78
4.5.1 Optimization Contribution . . . 79
4.5.2 Query Execution Time . . . 81
4.6 Related Work . . . 82

5 Conclusion and Future Work . . . 83

References . . . 85

LIST OF FIGURES

3.1 Performance of Stratified Query vs. RaSQL . . . 21
3.2 RaSQL Query Plan - (a) Clique (b) Physical . . . 33
3.3 Dataflow of DSN iterations . . . 39
3.4 Shuffle-Hash Join vs. Sort-Merge Join . . . 42
3.5 Dataflow of optimized DSN iterations . . . 44
3.6 Effect of Stage Combination . . . 45
3.7 Effect of Decomposition and Compression . . . 46
3.8 Effect of Code Generation . . . 47
3.9 REACH Experiments on RMAT-n graphs . . . 52
3.10 CC Experiments on RMAT-n graphs . . . 52
3.11 SSSP Experiments on RMAT-n graphs . . . 53
3.12 Systems performance comparison of REACH, CC and SSSP on real graphs. . . . 53
3.13 Performance comparison on Delivery, Management and MLM queries. . . . 55
3.14 Scaling-out Cluster Size . . . 56

4.1 Price stream (rising period in bold text) . . . 60
4.2 CEP system's ordering vs. User expected ordering . . . 61
4.3 Simplified syntax of WSP . . . 63
4.4 Example of a simple *weighted automaton*. . . . 70
4.5 Edge Generation Cases . . . 72
4.6 Example of Multiple Edges Elimination[1] . . . 73
4.7 Weighted automaton of Query Example 2 . . . 75
4.8 Sample matched results[2] of Figure 4.7 automaton . . . 75
4.9 Cache Optimization to reduce backtracks . . . 77
4.10 Constraints Transformation in Example 2 . . . 78
4.11 Query Types . . . 79
4.12 Optimization Contribution . . . 80
4.13 Performance Results . . . 81

LIST OF TABLES

3.1 Parameters of Synthetic Graphs 50
3.2 Parameters of Real World Graphs 51
3.3 CC Benchmark (in seconds) 56

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Professor Carlo Zaniolo for his guidance, support, and trust during my PhD study. The valuable suggestions and tremendous freedom that he gave me in the past five years, are a great memory for me and will always be an invaluable treasure for my life. I would also like to thank Professor Junghoo Cho, Professor Todd D. Millstein, and Professor Yingnian Wu for their time and help on my research.

Then I wish to thank the faculty members and students in ScAi lab for their support and encouragement, especially my frequent collaborators: Yugo Watanabe and Jin Wang. Moreover, I would like to thank Shi Gao and Kai Zeng, who gave me generous help and suggestions in my first two years of Ph.D. study. We collaborated on the RDF-TX and ABS projects, in which I learned a lot about research skills. I also want to thank Mohan Yang and Alexander Shkapsky, who made the wonderful BigDatalog system and offered tremendous support during my project. I want to thank all staff members in the Computer Science Department of UCLA, especially Joseph Brown, who has been such an excellent graduate student affairs officer with so much help in my Ph.D. study at the UCLA.

At last, I would like to thank my wife and my parents for their unconditional love. Any concrete words cannot describe my gratitude to the great support from my family members. Their support and understanding always made the difficult time easier for me, together making my dreams come true.

VITA

2012	Summer Exchange Student, CSST Program, UCLA.
2013	Bachelor of Engineering in Software Engineering, Fudan University.
2014 – 2018	Master of Science, Computer Science, UCLA.
2014 – 2019	Teaching Assistant, Computer Science Department, UCLA.
2014 – 2019	Research Assistant, Computer Science Department, UCLA.

PUBLICATIONS

Jiaqi Gu, Yugo H. Watanabe, William A. 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. SIGMOD, 2019.

Shi Gao, **Jiaqi Gu** and Carlo Zaniolo. RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases. EDBT, 2016.

Jiaqi Gu, Jin Wang and Carlo Zaniolo: Ranking support for matched patterns over complex event streams: The CEPR system. ICDE, 2016.

Shi Gao, Muhao Chen, Maurizio Atzori, **Jiaqi Gu** and Carlo Zaniolo. SPARQLT and its User-Friendly Interface for Managing and Querying the history of RDF knowledge base. International Semantic Web Conference, 2015.

Kai Zeng, Shi Gao, **Jiaqi Gu**, Barzan Mozafari and Carlo Zaniolo. ABS: the Analytical Bootstrap System for Fast Error Estimation in Approximate Query Processing. SIGMOD, 2014.

CHAPTER 1

Introduction

With the rise of Big Data and AI, there is growing interest in applications of large-scale data analytics. The sheer volume, variety and velocity of Big Data bring major challenges to current data analytics architectures. In fact modern data analytics systems must be able to process huge amounts of data in a very limited time period, i.e. achieving high throughput with a relative low latency, while still capable for handling diverse workloads. Moreover, the recent advances in AI call for languages and systems that can express complex application logics in machine learning and data mining, with a moderate learning curve that can be widely accepted by users.

As these different applications demand efficient support for complex analytics on Big Data, researchers in both academia and industry have proposed and built a number of large-scale distributed data processing systems and languages, such as Google MapReduce [ABC11], Microsoft DryadLinq [YIF08], Apache Spark [ZCF10], SAP HANA [FML12] and so on. These systems have been deployed on thousands of machines to extract the valuable information hidden in the Big Data, in application domains such as search engines, user behavior analysis, ads analytics, network analysis, fraud detection, etc., bringing millions of dollars' profits to the companies.

These systems support general data-parallel computations via Map-Reduce and similar techniques, which greatly facilitate users conducting certain kinds of tasks. However, complex data analytics call for expressive languages that can support complex business logics. Recently, more attention was paid to the classical declarative RDBMS query language – SQL,

given its ability to express common data analytics logics in concise declarative syntax. Indeed in recent years, we have seen a growth of popularity for SQL, which was adopted in various systems, including Apache Hive [TSJ10], Facebook Presto [pre] and Microsoft Scope [ZBW12]. Even systems, such as Apache Spark [AXL15], Apache Kafka [KNR11] and Google Spanner [BBB17], which were not initially targeted for relational processing are now actively building their SQL interfaces. The continuing popularity of SQL is hardly surprising given the huge benefits it offers over other languages for many Big Data applications. The main benefits include portability, performance and scalability via data parallelism that have been achieved through decades of research and industrial developments.

However, the expressive power of SQL limits its further adoption in specific data analytics domains, such as graph analytics and data mining. To overcome this limitation, ad-hoc systems and languages aimed to address these issues thrived in recent years. For example, more than twelve graph languages are designed and developed on various platforms, such as Cypher for Neo4j [FGG18], TitanDB[tit] and PowerGraph [GLG12], to support common graph analytical tasks such as shortest paths and connected components on large-scale distributed graphs. However, these approaches are hardly successful because (1) these domain-specific languages are not designed to solve general problems, thus become hard to inter-operate with other existing data analytical languages such as SQL, not to mention their inability to generate a widely accepted language standard; (2) their system implementations are highly influenced by the language design, whereby most of these systems are not as mature or efficient as the widely used data-parallel platforms such as Hive [TSJ10], MapReduce [DG08] or Apache Spark [ZCF10], not to mention the traditional RDBMS such as MySQL, Oracle and SQL Server with sophisticated optimization techniques built in.

A consensus is gradually emerging that it is urgent to devise more powerful query languages that can express complex analytical logics and are also conducive to very efficient distributed implementations. For example, to support expressing iterative computations, recursive structures need to be introduced into SQL. Another important requirement is the need to express pattern matching queries to support the analysis of time series and other

sequential data.

In response to these needs, the SQL standard introduced the recursive Common Table Expression (CTE), which is highly influenced by concepts and techniques developed by research work on DATALOG recursion. Also, to support queries on sequential data, the change proposal named SQL-MR (Match-Recognize) [ZWC07] was put forth by major DBMS vendors, extending the work done at UCLA by the SQL-TS project [SZZ01]. The research work done in academia has influenced on the continuous evolution of SQL standards, which underscores the significance of this line of research.

Several years of query language research have shown that aggregates in recursion are needed to support a wide range of analytics and big data applications, such as graph queries and data mining algorithms. In fact, aggregates in recursion enable the concise expression and efficient execution of powerful algorithms that cannot be supported efficiently, or cannot be supported at all, by queries that are stratified w.r.t. negation and aggregates [MSZ13]. However, stratification is required by the non-monotonic nature of common aggregates, such as min, max, count and sum, which compromise the declarative fixpoint semantics of recursive queries. This is a very difficult problem since the early days of DATALOG [KS91, GGZ91, RS92], a widely researched recursive query language. These approaches sought to achieve both (i) a formal declarative semantics for deterministic queries using the basic SQL aggregates, min, max, count and sum, in recursion and (ii) their efficient implementation by extending techniques of the early DATALOG systems [MUG86, COK87, SR92, VRK94, AOT03]. However, these approaches incurred problems such as those described by Van Gelder [Van93], or required more powerful semantics, such as answer-set semantics, that introduces higher levels of computational complexity [PDB07, SP07, SW10, FPL11, GZ14].

Fortunately, a novel approach based on the concept of Pre-Mappability (PreM) of constraints, which extends the declarative fixpoint semantics of Horn Clauses, was recently proposed by [ZYI18]. This provides a simple criterion that allows the optimizer to push constraints into recursion. Moreover, users can utilize it to write recursive queries with aggregates in recursion, with the guarantee that a formal fixpoint semantics holds.

Contribution Using recent advances in the theory of recursive queries, we proposed and demonstrated extensions to the SQL language and their efficient implementation, which provide effective answers to the challenges of supporting complex data analytics in the Big Data era. Thus, this dissertation makes significant contributions in the following areas.

Firstly, we propose a new recursive query language, RaSQL, an extension of current SQL standard that supports aggregates in recursion. The rigorous fixpoint semantics of the RaSQL language is provided by the PreM property [ZYD17]. The extension enables the expression of many algorithms in SQL, including but not limited to the common graph algorithms such as Single-Source-Shortest-Paths, Connected Components and PageRanks, that are widely used in complex data analytics. We implement the RaSQL on top of Apache Spark, with extensions made to the Spark SQL compiler and execution engine, achieving superior performance compared to most state-of-the-art systems supporting recursive computation, such as Apache Giraph, GraphX and Myria.

Secondly, we propose a new weighted search pattern language, WSP, which extends the SQL-TS [SZZ01] language, with simple constructs for specifying complex sequential patterns into SQL. The WSP language pushes forward the semantics in defining the ranking among searched patterns. By adoption the recent advances in weighted automata [FFG06], the implementation of the WSP language achieves a very good runtime performance on the single machine in processing millions of event tuples.

Outline The rest of this dissertation is organized as follows. In Chapter 2, we provide the background knowledge about Datalog, recursive SQL, Apache Spark and Spark SQL. In Chapter 3, we introduce the RaSQL language with its efficient implementation on top of Apache Spark. In Chapter 4, we provide the WSP language and its implementation on the single machine. We draw the conclusion and discuss the future work in Chapter 5.

CHAPTER 2

Background

2.1 Datalog

The recursive query originates from DATALOG [MSZ13], which is a declarative programming language and is often used as a query language for deductive databases. DATALOG has a succinct syntax and naturally supports recursive queries. DATALOG is getting increasing popularity in recent years as its new applications in information extraction, networking, program analysis and cloud computing [SYI16] due to its declarative nature in helping the expression of complex analytical logics.

A DATALOG program is composed of a finite set of rules. A *rule* r is denoted as $h \leftarrow b_1, \ldots, b_n$, where h is the *head* of r, and $b_1, \ldots, b_n$ is the *body* of r. The head h and b_i are *literals* having the form $p_i(t_1, \ldots, t_j)$, where p_i is a *predicate*, and $t_1, \ldots, t_j$ are *terms* that can be *constants*, *variables*, or *functions*. A *constant* can be a number, a string or any pre-defined value, and a *function* is a pre-defined mapping on the given variable inputs. The commas separating literals in the body represent logical conjunctions (AND). Disjuctions (OR) are represents by multiple rules with the same head predicate name. A *query* indicates the desired predicate to evaluate. Below is the Transitive Closure (TC) query in DATALOG:

Program 1. *Transitive Closure*

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

We have two rules in this program, *r1* and *r2*. Rule *r1* is called a *base rule* as its body

only refers to the existing relations in the database. Here, *r1* derives tuples in *tc* by simply including all tuples from the relation `edge`, which means that all directed edges in the graph are included as initial transitive closures. Rule *r2* is called a *recursive rule* since it refers to itself in its body. The program can be computed using a bottom-up evaluation: The rule *r1* is applied first, and then the rule *r2* is applied by recursively joining the previous `tc` tuples with the `edge` tuples until no more new tuples are generated, which defines its formal least fixpoint semantics as introduced next.

Fixpoint Semantics The semantics of the DATALOG program P are defined as the declarative least-fixpoint semantics of Horn Clauses, where specifically, in terms of the Immediate Consequence Operator (ICO) for P, denoted as $T_p(I)$, where I is any Herbrand interpretation of P. In practice, I can be informally viewed as a set of variable assignments that make the given DATALOG program valid. Similarly, the ICO can be viewed as an application (a single derivation) of the DATALOG rules on I. For a recursive query without negation, $T_p(I)$ is a monotonic continuous mapping in the lattice of set-containment to which the interpretation I belongs. As illustrated in [ZYD17], following well-known properties hold:

1. A unique minimal (w.r.t. set-containment) solution of the equation $I = T_p(I)$ always exists and it is known as the least-fixpoint of T_p, denoted $lfp(T_p)$; $lfp(T_p)$ defines the formal declarative semantics of P.

2. For an immediate consequence operator T, with ω being the first infinite ordinal, $T^{\uparrow\omega}(\emptyset)$ is defined by letting $T^{\uparrow 0}(\emptyset) = \emptyset$, 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. The *fixpoint iteration* $T_P^{\uparrow\omega}(\emptyset)$ defines the operational semantics of our program. In most applications of practical interest, the iteration converges to the final value in a finite number of steps, and it can be stopped at the first integer $n+1$ where, $T_P^{\uparrow n+1}(\emptyset) = T_P^{\uparrow n}(\emptyset)$.

For positive DATALOG programs, we have the operational and declarative semantics coincide. Recent literature [MPR90] shows that the ability to use aggregates in recursion will largely extend the expressiveness of the language. However, problems arise if we want to use basic

aggregates such as **min**, **max**, **count** and **sum** in recursion, especially when we want to push the non-monotonic logic such as aggregates or lower/upper-bound constraints into the DATALOG recursion, which is essential for efficient DATALOG evaluation optimizations such as Semi-Naïve evaluation. Fortunately, it was recently shown [ZYD17, CDI18] that recursive queries with aggregates that satisfy a *Pre-Mappability* (PreM) condition produce the same results as the stratified program, which means the declarative least-fixpoint semantics of Horn Clauses can be safely extended to basic SQL aggregates, along with the bottom-up evaluation techniques used by many existing DATALOG systems.

Datalog Evaluation Following the definition of the fixpoint semantics, we present the basic DATALOG evaluation technique called *naive evaluation* in Algorithm 1 for TC example.

Algorithm 1 Naive Evaluation of TC

1: $\texttt{tc} := \emptyset$
2: $\texttt{tc}' := \texttt{edge}(\texttt{X}, \texttt{Y})$
3: **DoWhile**
4: $\quad \texttt{tc} := \texttt{tc} \cup \texttt{tc}'$
5: $\quad \texttt{tc}' := \pi_{\texttt{X},\texttt{Y}}(\texttt{tc}(\texttt{X}, \texttt{Z}) \bowtie \texttt{edge}(\texttt{Z}, \texttt{Y}))$
6: **EndDoWhile**$(\texttt{tc}' \neq \texttt{tc})$
7: **return** tc

The naive evaluation method outlines the general procedure to evaluate a recursive query[1]. Here we use the example of computing Transitive Closures (TC) to describe this approach. First, the *base rule* is executed and the new tuples $\texttt{tc}'$ is produced using the edge relation (line 1-2). Second, in the start of each recursive iteration, the tc relation includes tuples generated from the previous iteration by union (line 4). Then the *recursive rule* joins tc tuples with tuples from the base relation edge to produce the new tuples $\texttt{tc}'$ (line 5). This process is executed iteratively until the tc relation no longer changes, i.e. a fixpoint is reached (line 6). Finally, tuples in tc relation are returned.

Though any recursive query language using fixpoint semantics conceptually evaluates the

[1]Language that adopts bottom-up evaluation and fixpoint semantics

same as the naive evaluation, this approach is seldomly used in real implementations because it will inefficiently re-produce the existing results in every iteration as all previous generated tuples in the recursive relation are participating in the derivation of new tuples. An improved method called Semi-Naïve evaluation is presented in section 3.6.

2.2 Recursive SQL

SQL:99 is the first SQL standard that introduces the recursive query. The design of SQL:99 recursion is largely influenced by Datalog. In order to refer itself within a recursive relation definition, the recursion is implemented using common table expressions (CTEs).

A CTE is a named temporary view within a SQL statement that is retained only for the duration of that statement. In SQL:99, a CTE view definition can contain references to itself, i.e. a recursive CTE. The *transitive closure* query written in SQL:99 CTE is shown in Program 2.

```
WITH recursive tc (X, Y) AS
  (SELECT X, Y FROM edge
   UNION ALL
  (SELECT tc.X, edge.Y FROM tc, edge
   WHERE tc.Y = edge.X)
SELECT * FROM tc
```

Program 2: TC query expressed as a recursive CTE

The recursive CTE construct starts with the keyword `WITH`, followed by a sequence of recursive *view definitions*. The definition starts with the optional keyword `recursive`, followed by the view schema (name and columns). The view is materialized by the *definition query* which is a *union* of sub-queries.

Similar to a valid recursive Datalog program, the recursive relation defined in SQL also

requires to incorporate *base cases* and *recursive cases*. Indeed, sub-queries that do not reference to the defining relation are treated as *base cases*, while others are *recursive cases*. In Program 2, the first sub-query is the *base case* while the second is the *recursive case*. Despite the syntax differences, Program 1 and 2 share nearly the identical semantics, and the evaluation of Program 2 is also the same as described in Algorithm 1.

SQL:99 standard has already been supported in most commercial database systems such as DB2, Microsoft SQL Server, Oracle, and PostgreSQL. For most implementations, they adopt the same fixpoint semantics and evaluation methods as the DATALOG for recursive SQLs, thus we skip the redundant materials. Similarly, new semantics needs to be proposed if *aggregates* or other non-monotonic reasoning needs to be supported in recursion.

2.3 Apache Spark and Spark SQL

Apache Spark [ZCF10] is a popular large-scale distributed computation framework with Resilient Distributed Dataset (RDD) as its core abstraction. The RDD model abstracts a distributed partitioned dataset and a series of coarse-grained operations, such as *map*, *filter*, *join* takes effect on it. Transformations are performed lazily, i.e. they will not be submitted for execution until an action such as *count* is called. Once a job is submitted, the scheduler will divide its RDD transformation DAG into a series of *stages*. The transformations that can be pipelined (e.g., a series of *map* operations) are grouped into a single stage. Between stages, Spark shuffles the dataset to repartition it among the workers. When all dependent stages of a stage complete execution, this stage will be executed. In this case, the scheduler creates a set of tasks (i.e., execution units) consisting of one task per input partition, and launches the tasks on workers.

Spark SQL [AXL15] is the relational data processing module on top of Spark. It includes the full-fledged SQL parser, analyzer, optimizer and planner which compile an SQL query into a DAG of RDD transformations. Thus Spark SQL allows users to express complex analytics logics in high-level Dataset or SQL APIs without the knowledge of RDDs. Spark

SQL supports the SQL:2003 standard but without recursive CTE.

Partitioning Relational Dataset The dataset processed in a distributed system, such as Apache Spark, is often *partitioned* and distributed on multiple nodes. In Hadoop or Spark, each executor only processes a single partition at one time. We discuss the *partitioning scheme* for a relational dataset $\mathbf{R}$.

A *partition function* decides which partition a given tuple in $\mathbf{R}$ belongs to, defined as follows: Let $\mathbf{C} = (C_{i_1}, \ldots, C_{i_k}) \subseteq \mathbf{R}$ be a subset of attributes of relation $\mathbf{R}$, and $\mathbf{D} = D_{i_1} \times D_{i_2} \ldots, \times D_{i_k}$ be the domain of values in attribute set $\mathbf{C}$. A partition function h over the *partition key* $\mathbf{C}$ is defined as a hash function which maps any value $\mathbf{d} = (d_{i_1}, \ldots, d_{i_k}) \in \mathbf{D}$ to a partition id (pid), i.e., $pid = h(\mathbf{d}), h : \mathbf{D} \rightarrow \{1, \ldots, n\}$.

A dataset needs to be *repartitioned* if it does not satisfy the required partitioning of a specific operation. In this case, a *shuffle* operation will be involved which moves data between different workers/executors to make sure the dataset is re-arranged in the desired way. In the classical word count example [wor] in MapReduce, the Reduce stage requires its input to be partitioned on *word* so that all map outputs belong to the same *word* can be collected into the same partition in order to correctly generate the total count of each word. As a result, all map outputs are *shuffled* during this *repartition* process. *Shuffle* operation is very expensive as it often incurs large IO costs among all workers, thus we want to avoid it as much as possible.

CHAPTER 3

RaSQL: Greater Power and Performance with Recursive-aggregate-SQL

In this chapter, we introduce our extension to the current SQL standard, the Recursive-aggregate-SQL (RaSQL) language. Thanks to it, we can express very powerful queries and declarative algorithms, such as classical graph algorithms and data mining algorithms in pure SQL way. A novel compiler implementation allows RaSQL to map declarative queries into one basic fixpoint operator supporting aggregates in recursive queries. A fully optimized implementation of this fixpoint operator over multiple platforms leads to superior performance, scalability and portability. We will then present our RaSQL system, which extends Spark SQL with the before-mentioned new constructs and implementation techniques, matches and often surpasses the performance of other graph or distributed computation systems, including Apache Giraph, GraphX and Myria.

3.1 Introduction

An exploding number of Big Data and AI applications demand efficient support for complex analytics on large and diverse workloads. To address this challenge, researchers in both academia and industry have built a number of large-scale distributed data processing systems relying on various language interfaces [MAB10, WBB17, YIF08, ZCF10]. Besides those that propose new languages, we find several of these, such as Apache Hive [TSJ10], Facebook Presto [pre] and Microsoft Scope [ZBW12], all choose SQL or its dialects as their language.

Even systems which were not targeted at relational processing initially, are now actively building their SQL interfaces, such as Apache Spark [AXL15], Apache Kafka [KNR11] and Google Spanner [BBB17].

The continuing popularity of SQL is hardly surprising given the huge benefits it offers over other languages for many big-data applications, including its portability, performance and scalability via data parallelism that has been achieved through decades of research and industrial developments. Yet, as the field is witnessing the emergence of many data management systems supporting new applications, such as graph and AI applications, a consensus is growing that devising simple extensions of SQL that will allow RDBMS to efficiently support these applications represents a vital research problem. Thus, while projects such as [OPV14] focus on enabling SQL to support the richer data structures and representations provided by JSON, here, we focus on extending the expressive power of the language.

The potential for the greater expressive power of recursive queries has provided a key motivation for Datalog research [KS91, MPR90, MSZ13]. Inspired by it, the SQL:99 standard introduced the Recursive Common Table Expression (CTE), which allows recursive queries such as Transitive Closure (TC). However, in those extensions, the use of negation and aggregates in the recursive query proper was disallowed due to the concern that their non-monotonic nature would compromise the least-fixpoint semantics of the queries. Therefore, the current SQL standard merely supports queries that are *stratified* w.r.t. aggregates and negation, leading to computations where the recursive queries complete before aggregates and negation are applied to the results that they produce.

Fortunately, it was recently shown [ZYD17, CDI18] that queries with aggregates-in-recursion that satisfy a *Pre-Mappability* (PreM) condition produce the same results as the *stratified* program; thus they are semantically equivalent but evaluate much more efficiently than their stratified counterparts. Furthermore, recursive queries that do not use negation but aggregates satisfying PreM can express many advanced queries of interest, such as those discussed here and in [CDI18].

In this chapter, we present the RaSQL[1] language and system that extends the current SQL standard to support aggregates-in-recursion. RaSQL is able to express a broad range of data analytics queries, especially those commonly used in graph algorithms and data mining, in a fully declarative fashion. This quantum leap in expressive power is achieved by enabling the use of the aggregates such as *min*, *max*, *count* and *sum* in the CTE in the SQL standard.

Furthermore, the RaSQL system compiles the recursive construct into a highly optimized *fixpoint operator*, with seamless integration to other SQL operators such as joins or filters. Because of the aggregate-in-recursion optimization brought by PreM and other improvements discussed in this chapter, the performance of RaSQL system implemented on top of Apache Spark, matches and often surpasses that of other systems, including Apache Giraph, GraphX and Myria as shown in our experiments (Section 3.8).

The main purpose of our experiments is not to explore how fast we can run on the most advanced hardware, or claim that the RaSQL implementation is fundamentally faster than some other system, but rather to demonstrate that a general recursive query engine can be optimized to achieve the competitive performance of special-purpose graph systems.

Contributions. We make the following contributions:

- The RaSQL language which extends the power of recursion of the current SQL standards, and enables expression of a broad range of applications in fully declarative ways.
- A series of compiler implementation techniques to map the recursive query into a simple *fixpoint operator*, which is amenable to a variety of optimizations.
- Distributed iterative computation optimizations specifically focused on recursive plans, i.e. the fixpoint evaluation process, and various improvements in job scheduling, data broadcasting and shuffling.
- A large-scale evaluation of the RaSQL implementation on real graphs and complex ana-

[1]RaSQL, pronounced 'Rascal', is an acronym for: Recursive-aggregate-SQL.

lytics tasks comparing our system with other special-purpose graph systems.

Outline. In Section 3.2, we introduce the syntax and semantics of the RaSQL language. Section 3.3 describes the PreM property which is the foundation for making possible the efficient evaluation of aggregates in recursion. Section 3.4 gives examples of several important algorithms that can be succinctly expressed in RaSQL. Section 3.5 shows how RaSQL queries are compiled into recursive physical plans, i.e. the *fixpoint operator*, to execute on Apache Spark. Section 3.6 introduces the *fixpoint operator* execution, specifically the distributed semi-naive evaluation process. Section 3.7 presents more optimization techniques, including stage combination, decomposed plan evaluation and code generation. Section 3.8 presents experimental results, with comparisons of our system to other distributed iterative computation systems. Section 3.9 reviews related works.

3.2 The RaSQL Language

RaSQL is a new query language which is a superset of the current SQL:99 standard, thus all SQL:99 language features are supported in RaSQL. Beyond that, RaSQL makes important extensions to the standard's Common Table Expressions (CTEs) to allow the use of basic aggregates in recursion, with its semantics guaranteed by the PreM property.

Syntax As demonstrated below, the CTE construct starts with the keyword "WITH", followed by a sequence of recursive view definitions. Each recursive view definition starts with the optional keyword "recursive", followed by the view schema definition (name and columns). The view content is defined by a *union* of sub-queries: each sub-query is either a *base* case, i.e. its FROM clause does not refer to any recursive CTE, or a *recursive* case, i.e. its FROM clause may refer to one or more recursive CTEs.

```
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

RaSQL extends the CTE construct for recursive queries by allowing the use of the basic aggregates such as *min*, *max*, *sum*, *count* in recursive view definition. For aggregates used in the CTE, RaSQL adopts the *implicit group by* rule, i.e. no explicit group by clause is required in each sub-query, and all columns except the aggregates will be treated as the group by columns. This design greatly reduces the boilerplate code and improves the readability. Moreover, it conveys the message that the aggregate function is applied to all the given column values produced during the recursive computation.

Example The Bill of Materials (BOM) [bom] is an important recursive query application in SQL:99, and gets widely used in many business environments. We use this classical example to illustrate the detailed syntax and semantics of RaSQL and the PreM property. In our simplified version of BOM, we have the following two base relations:

```
assbl(Part, SPart).      basic(Part, Days).
```

The `assbl` table describes the assembling relationships between an item and its immediate subparts. Not all items are assembled: basic parts are purchased directly from external suppliers and will be delivered in a certain number of days, which is described in the `basic` table. Without loss of generality, we assume a part will be ready the very same day when its last subpart arrives. Thus, the number of days required for the an item to be ready is the *maximum* day when each of its subparts is delivered. In SQL:99, the BOM query can be expressed as follows:

```
WITH recursive waitfor(Part, Days) AS
  (SELECT Part, Days FROM basic) UNION
  (SELECT assbl.Part, waitfor.Days
   FROM assbl, waitfor
```

```
   WHERE assbl.Spart = waitfor.Part)
SELECT Part, max(Days) FROM waitfor GROUP BY Part
```

Q1: Days Till Delivery by a Stratified Program

RaSQL still supports this query, but also supports an equivalent and much more efficient version as below.

```
WITH recursive waitfor(Part, max() as Days) AS
  (SELECT Part, Days FROM basic) UNION
  (SELECT assbl.Part, waitfor.Days
   FROM assbl, waitfor
   WHERE assbl.Spart = waitfor.Part)
SELECT Part, Days FROM waitfor
```

Q2: The Equivalent Endo-Max Program

Note that from a syntax point of view, the RaSQL query only makes a small change to replace the *stratified max*[2] used in $Q1$ by the *max* aggregate in the recursive CTE head of $Q2$. However, it is non-trivial to show that $Q1$ and $Q2$ are equivalent semantically and evaluate to the same result, which is discussed below.

Semantics The semantics for query $Q1$ and $Q2$ are defined by *naive fixpoint evaluation* shown in Algorithm 1 and 2, where the following abbreviations are used:

$\texttt{wf}=\texttt{waitfor}$ $\quad \pi=\pi_{\texttt{assbl.Part,wf.Days}}$

$\bowtie=\bowtie_{\texttt{assbl.SPart=wf.SPart}}$ $\quad \texttt{max}=\ _{\texttt{Part}}\texttt{max}_{\texttt{Days}}$

We denote the *Relational Algebra* (RA) expression used in line 5 of Algorithm 1 as $\texttt{T}$, which derives the new $\texttt{wf}'$ from the old $\texttt{wf}$. The operators used by $\texttt{T}$ are union, join and project, which are monotonic and guarantees that when $\texttt{wf}' = \texttt{wf}$, we obtain the *least fixpoint* solution of equation $\texttt{wf} = \texttt{T}(\texttt{wf})$, which defines the formal semantics of the Recursive CTE $Q1$. The

[2]stratified aggregate means it can only be applied to the result of recursive evaluation, not *within* the recursive evaluation process.

Algorithm 2 Naive Evaluation of waitfor (wf) in Q1

1: $\texttt{wf} \leftarrow \emptyset$

2: $\texttt{wf}' \leftarrow \texttt{basic}(\texttt{Part}, \texttt{Days})$

3: **repeat**

4: $\quad \texttt{wf} \leftarrow \texttt{wf} \cup \texttt{wf}'$

5: $\quad \texttt{wf}' \leftarrow \pi(\texttt{assbl}(\texttt{Part}, \texttt{SPart}) \bowtie \texttt{wf}(\texttt{SPart}, \texttt{Days})))$

6: **until** $\texttt{wf}' = \texttt{wf}$

7: **return** $\texttt{max}(\texttt{wf})$

Algorithm 3 Naive Evaluation of waitfor (wf) in Q2

1: $\texttt{wf} \leftarrow \emptyset$

2: $\texttt{wf}' \leftarrow \texttt{max}(\texttt{basic}(\texttt{Part}, \texttt{Days}))$

3: **repeat**

4: $\quad \texttt{wf} \leftarrow \texttt{wf} \cup \texttt{wf}'$

5: $\quad \texttt{wf}' \leftarrow \texttt{max}(\pi(\texttt{assbl}(\texttt{Part}, \texttt{SPart}) \bowtie \texttt{wf}(\texttt{SPart}, \texttt{Days})))$

6: **until** $(\texttt{wf}' = \texttt{wf})$

7: **return** $\texttt{wf}$

last line in $Q1$ contains a non-monotonic `max` aggregate which is applied to `wf` when the *fixpoint* $\texttt{wf} = \texttt{T}(\texttt{wf})$ is reached. This computation is performed at line 7 of Algorithm 1, which computes the perfect-model for query $Q1$ that is stratified w.r.t. *max* [ZCF97].

Algorithm 2 shows the naive-fixpoint evaluation for $Q2$. The *max* aggregate has been moved from the final statement in Q1 to the head of the Recursive CTE in $Q2$. Thus the `max` aggregate which was in line 7 of Algorithm 1 has now been moved to line 1 and 5 of Algorithm 2— i.e., lines that specify the initial and iterative computation of `wf`.

This example illustrates that supporting aggregates in the `WITH recursive` clause requires only simple extensions at the syntax level, and techniques such as semi-naive fixpoint and magic sets also require simple extensions when aggregates are allowed in recursion [CDI18]. However, aggregates in recursion raise major semantics issues caused by their non-monotonic nature, which have been the focus of many years of research described in Section 3.9. Fortunately, the recent introduction of the PreM property has enabled much

progress [ZYD17, ZYI18] on this problem. In fact, the PreM holds for this example which indicates $Q1$ and $Q2$ will produce the same results (concrete semantics) and $Q2$ has a minimal fixpoint semantics that is equivalent to the perfect model semantics of $Q1$ (abstract semantics) [ZYD17].

3.3 The PreM property

If $T(R_1, \ldots R_k)$ is a function defined by RA we say that a constraint γ, is PreM to $T(R_1, \ldots, R_k)$ when the following property holds:

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

For instance, if T is *union* and γ is `max`, then we have:

$$max(R_1 \cup \ldots \cup R_k) = max(max(R_1) \cup \ldots \cup max(R_k)).$$

PreM for *union* has long been recognized and used as a cornerstone for parallel databases and MapReduce. Moreover the PreM property also holds for *join* and other operators, which provides a general solution to the semantic problem of having aggregates in recursive queries.

Consider the RA expression T used in line 5 of Algorithm 2, which computes the new value of $\texttt{wf}'$ from its old value `wf`. The PreM property holds for *max* if $\texttt{max}(\texttt{T}(\texttt{wf})) = \texttt{max}(\texttt{T}(\texttt{max}(\texttt{wf})))$, i.e. if the following relational expressions are equivalent:

$$\texttt{max}(\pi(\texttt{assbl}(\texttt{Part}, \texttt{SPart}) \bowtie \texttt{wf}(\texttt{SPart}, \texttt{Days}))) =$$

$$\texttt{max}(\pi(\texttt{assbl}(\texttt{Part}, \texttt{SPart}) \bowtie \texttt{max}(\texttt{wf}(\texttt{SPart}, \texttt{Days}))))$$

This PreM equality states that the *max* waitfor days for a part is the *max* of the max of days for each of its subparts. For simple queries such as this, proving the PreM property is straightforward. Powerful techniques are available for proving that complex expressions of RA and arithmetic operators are PreM w.r.t. extrema and other aggregates [ZYI18].

With the PreM property holding, we can now observe that $\texttt{max}(\texttt{wf}(\texttt{Part}, \texttt{Days}))$ simply denotes the application of *max* to the previous step in the computation of Algorithm 1, by applying the join and the PreM rule interchangeably, we can derive Algorithm 2 from Algorithm 1:

$$
\begin{aligned}
&\texttt{max}(\texttt{wf}_\texttt{n}(\texttt{Part}, \texttt{Days})) \\
=\ &\texttt{max}(\pi(\texttt{assbl}(\texttt{Part}, \texttt{SPart}) \bowtie \texttt{wf}_{\texttt{n}-1}(\texttt{SPart}, \texttt{Days}))) \\
=\ &\texttt{max}(\pi(\texttt{assbl}(\texttt{Part}, \texttt{SPart}) \bowtie \texttt{max}(\texttt{wf}_{\texttt{n}-1}(\texttt{SPart}, \texttt{Days}))) \\
=\ &\texttt{max}(\pi(\texttt{assbl}(\texttt{Part}, \texttt{SPart}) \bowtie \\
&\texttt{max}(\pi(\texttt{assbl}(\texttt{Part}, \texttt{SPart}) \bowtie \texttt{max}(\texttt{wf}_{\texttt{n}-2}(\texttt{SPart}, \texttt{Days})))) \\
&\ldots \\
=\ &\texttt{max}(\pi(\texttt{assbl}(\texttt{Part}, \texttt{SPart}) \bowtie \ldots \\
&\texttt{max}(\pi(\texttt{assbl}(\texttt{Part}, \texttt{SPart}) \bowtie \texttt{max}(\texttt{basic}(\texttt{SPart}, \texttt{Days})))))
\end{aligned}
$$
[3]

Symmetrically, if we start from Algorithm 2, we see that $\texttt{max}(\texttt{wf}(\texttt{Part}, \texttt{Days}))$, i,e., the *max* applied to the previous step in the iteration, can be removed without changing the result. Therefore, starting at line 2, and repeating this reasoning for each successive step, we can remove each *max* from Algorithm 2, except the one at the last step, that will actually be applied at line 7. In other words, we find that PreM allows us to transform Algorithm 1 into Algorithm 2 and vice-versa, thus providing a robust proof of the equivalence of the operational semantics of Q1 and Q2. At the pure declarative level, the abstract semantics of Q1 and Q2 coincide, since the *minimal fixpoint* semantics for Q2 coincides with the perfect model of Q1 [ZYD17].

In addition to *min* and *max*, the PreM condition entails the use of *count* and *sum* in recursion. This is because, the *count* can be modeled as the *max* applied on the continuous *monotonic count* which can be freely used in recursion, and similar property hold for the *sum* of positive numbers [ZYD17, CDI18]. Thus for our BOM example, we can use a query similar to Q2 to efficiently compute the *count* of items used in an assembly, or to *sum* their costs.

[3] Due to limited space, union is omitted, given that PreM holds for union.

3.3.1 Programming with Aggregates in Recursion

The stratified version Q1 and the unstratified version Q2 are semantically equivalent. Therefore, in principle, either one could be employed by users to express their applications. However, there are three compelling reasons for RaSQL to adopt the unstratified syntax, and these outweigh the benefits of using internal query rewritings through the compiler.

A first reason for this conclusion is that the iteration implied by the fixpoint evaluation combined with *max* or *min* provides a natural high-level abstraction of the original procedural algorithms that users find very helpful in expressing complex algorithms in RaSQL. The same cannot be said for the stratified versions, particularly for queries such as SSSP[4] where cycles in the database will cause non-termination. Programmers who in the course of their past experience have developed an aversion to non-terminating loops will feel very uneasy about that query. The same is true for users who are not familiar with the concept of transfinite computations, a concept needed to understand the semantics of an SSSP query in the presence of cycles in the graph.

A second reason for preferring the unstratified versions of the queries is the semantics of *count* and *sum*. In fact, the semantics of a traditional *count* is defined by the application of *max* to the continuous count which is monotonic and can thus be used in recursion [MSZ13, ZYD17]. Therefore, stratified programs will have to use the *monotonic count* in recursion and then take the maximum at the next stratum. On the other hand, by allowing the use of count in recursion, RaSQL avoids the need to introduce a special *monotonic count* aggregate all together. Similar observations will also hold for *sum* (but not for *avg* since the ratio of *monotonic count* and *sum* is not *monotonic*).

A third reason is that the Q2 syntax clearly states to users that the optimization resulting from PreM will be applied in the course of the fixpoint computation which often exhibits much better performance than Q1–and there are even cases, such as SSSP with directed

[4]Single-Source-Shortest-Path (SSSP) algorithm, shown in Section 3.4

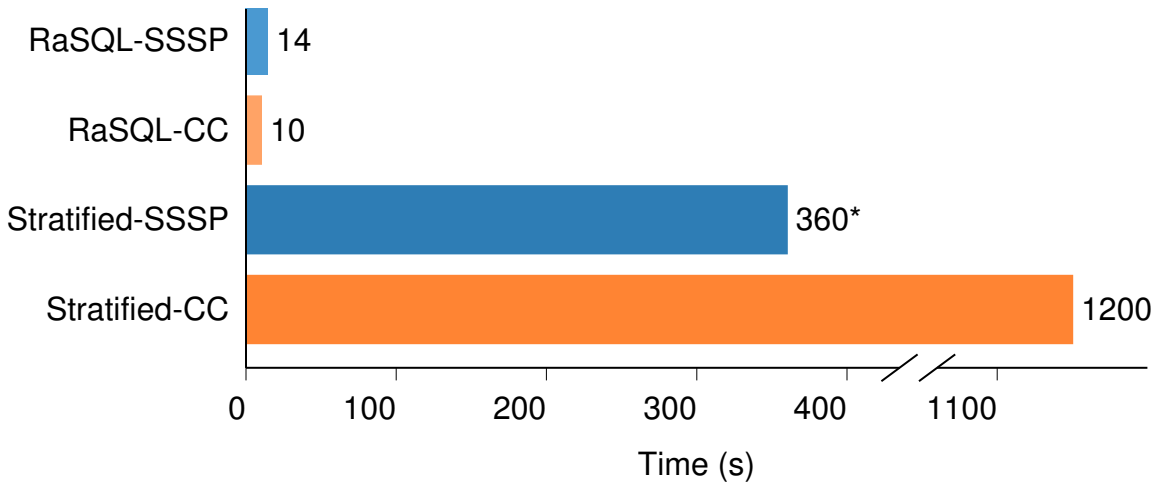


Figure 3.1: Performance of Stratified Query vs. RaSQL

cycles, where Q2 terminates but Q1 does not. Our experiments result in Figure 3.1 shows that a typical *unstratified* query runs orders of magnitude faster than their *stratified* counterparts on widely used graph applications such as CC and SSSP[5].

Of course, before replacing Q1 with the much more preferable Q2, users should ensure that PreM holds for their queries. This task can be performed using the proof techniques introduced in [ZYI18]. However, a formal proof is often an over-kill for people who are used to testing rather than proving the correctness of their queries. These users will instead want to test that PreM held at each step of the computations by which they tested the correctness of their programs. Thus we provide an auto-validation tool bundled with the RaSQL compiler, which helps validate arbitrary RaSQL query (Section 3.3.2). Moreover, we built a library of commonly used graph and data mining algorithms written in RaSQL and we proved that they satisfy PreM. These queries can be easily modified to adapt to a wider range of application scenarios.

[5]*Only the execution time for meaningful iterations is recorded for stratified-SSSP, as it will not terminate due to loops in the graph.

3.3.2 Testing PreM

When testing a recursive query, to check that it produces the intended result, the RaSQL system will also provide the users with a simple way to verify that the result satisfies the formal declarative semantics defined by the perfect model of the stratified version of the query. This is guaranteed by the following theorem.

Theorem 3.3.1. *If the PreM is satisfied at each step of the fixpoint computation, the perfect model for its stratified version of the query is obtained.*

Now, checking that PreM holds at each step of the fixpoint can be achieved by re-writing the original program into its PreM-checking version. For instance, Query G2 below shows the PreM checking version of our original APSP query G1.

```
WITH Recursive apsp(Src, Dst, min() AS Cost) AS
   (SELECT Src, Dst, Len FROM edge)
     UNION
   (SELECT apsp.Src, edge.Dst, apsp.Cost+edge.Len
    FROM apsp, edge WHERE apsp.Dst = edge.Src)
SELECT Src, Dst, Cost FROM apsp
```

Query G1: Original APSP Query

```
WITH Recursive all(Src, Dst, Cost) AS
   (SELECT Src, Dst, Len FROM edge)
     UNION
   (SELECT apsp.Src, edge.Dst, apsp.Cost+edge.Len
    FROM apsp, edge WHERE apsp.Dst = edge.Src),
Recursive apsp(Src, Dst, min() AS Cost) AS
   (SELECT Src, Dst, Len FROM edge)
     UNION
```

```
   (SELECT all.Src, edge.Dst, all.Cost+edge.Len
    FROM all, edge WHERE all.Dst = edge.Src)
SELECT Src, Dst, Cost FROM apsp
```

Query G2: PreM-checking version - APSP

Observe that G2 can be easily derived from G1, by introducing an additional recursive view, `all`, that provides the un-minimized counterpart of `apsp`. In fact, the statements computing `all` in Query G2 are those that compute `apsp` in G1, with the *min* constraint removed. Symmetrically, the statements computing `apsp` in Query G2 are obtained from G1 by replacing `apsp` by `all` in its recursive case. Therefore, `apsp` in Query G2 implements the computation $\gamma(T(I))$, whereas in the original Query G1 it was computed as $\gamma(T(\gamma(I)))$. As long as we verify that Query G1 and G2 produce the same `apsp` results at each step of the computation, we are confident that the PreM holds at each step and thus the query is correct.

The testing technique that we discussed for the APSP is quite general and any RaSQL programmer can use it to test PreM on their exo-min or endo-min queries. Furthermore, we are implementing a debugging tool called `GPtest` that automates this task for users: Similar to a step-by-step debugger, `GPtest` executes both the original and the PreM-checking version queries iteration by iteration, and will signal a failure of PreM as soon as a difference is detected, which calls for revisions of the original programs. We hope that this will help users quickly learn how to avoid violations of PreM when writing their queries.

While testing will be sufficient to satisfy many users, a formal proof of PreM is preferable in many situations, particularly when introducing advanced declarative algorithms written in RaSQL such as those published on our website [web]. Such proofs can be performed by users exploiting techniques similar to those described in [ZYI18], or by `GPtest` which is now being extended to automate the PreM-proving process.

3.4 RaSQL Examples

We provide a list of example queries written in RaSQL: Example 1-3 are classical graph queries; Example 4-8 demonstrate various application queries expressible in RaSQL.

Example 1 - Single-Source-Shortest-Path (SSSP):

```
Base tables: edge(Src: int, Dst: int, Cost: double)
WITH recursive path (Dst, min() AS Cost) AS
  (SELECT 1, 0)
   UNION
  (SELECT edge.Dst, path.Cost + edge.Cost
   FROM path, edge
   WHERE path.Dst = edge.Src)
SELECT Dst, Cost FROM path
```

The SSSP query computes shortest paths from a given source node to all other nodes in the graph. The weighted edges are stored in the base relation *edge*. The recursive relation *path* is initialized with a path of length 0 to the source node itself. The paths to other nodes are iteratively computed by joining existing paths with edges that start from the end of these paths. The *min* aggregate is used to select the shortest path.

Example 2 - Connected-Components (CC):

```
Base tables: edge(Src: int, Dst: int)
WITH recursive cc (Src, min() AS CmpId) AS
  (SELECT Src, Src FROM edge)
   UNION
  (SELECT edge.Dst, cc.CmpId FROM cc, edge
   WHERE cc.Src = edge.Src)
SELECT count(distinct cc.CmpId) FROM cc
```

The CC query finds all connected components in a graph. The idea of this algorithm is label propagation. Each node is attached a component id *CmpId* denoting which component it belongs to. Initially, it is assigned as its own *id*. During iterations, the *CmpId* of each node is updated as the minimal *CmpId* from its neighbors. The final result is calculated by counting the distinct number of *CmpId*s as all nodes within a single connected component will have the same (minimal) *CmpId* when the fixpoint is reached.

Example 3 - Count Paths:

```
Base tables: edge(Src: int, Dst: int)
WITH recursive cpaths (Dst, sum() AS Cnt) AS
  (SELECT 1, 1)
   UNION
  (SELECT edge.Dst, cpaths.Cnt FROM cpaths, edge
   WHERE cpaths.Dst = edge.Src)
SELECT Dst, Cnt FROM cpaths
```

The Count Paths query computes the number of paths from a given node to all nodes in a graph. The *cpaths* is initialized by 1, which is the count of paths from the start node to itself. The number of paths from the start node to another node is iteratively computed by adding up the path count from the start node to an intermediate node, which directly connects to the destination code.

Example 4 - Management:

```
Base tables: report(Emp: int, Mgr: int)
WITH recursive empCount (Mgr, count() AS Cnt) AS
  (SELECT report.Emp, 1 FROM report)
   UNION
  (SELECT report.Mgr, empCount.Cnt
   FROM empCount, report
   WHERE empCount.Mgr = report.Emp)
```

```
SELECT Mgr, Cnt FROM empCount
```

The Management query calculates the total number of employees that a manager directly and indirectly manages in a large corporation. The base relation *report* describes the relationship between an employee and his/her manager. In the base case, the employee count for everyone is initialized to 1. In the recursive case, the employee count of a manager is iteratively computed by adding up the employee count of his/her direct reporters.

Example 5 - MLM Bonus:

```
Base tables: sales(M: int, P: double)
             sponsor(M1: int, M2: int)
WITH recursive bonus(M, sum() as B) AS
  (SELECT M, P*0.1 FROM sales)
   UNION
  (SELECT sponsor.M1, bonus.B*0.5 FROM bonus, sponsor
   WHERE bonus.M = sponsor.M2)
SELECT M, B FROM bonus
```

The MLM Bonus query calculates the bonuses that a company using a multi-level marketing model [mlm] needs to pay its members. The salesmen in such a company form a pyramid hierarchy, where new members are recruited into the company by old members (sponsors) and get products from their sponsors. To encourage its members to sell more products, the company rewards them through bonuses. A member's bonus is not only based on his/her own personal sales, but also the sales of each member in the pyramid network that he/she directly and indirectly sponsored.

The *sales* relation describes the gross profit P that each member makes, and the *sponsor* relation shows the sponsor relationship between two members. In the base case, it calculates the bonus that a member earns through the products that they sold by themselves. In the recursive case, it calculate the bonus that derived from the sales of each member that he/she

directly/indirectly sponsors.

Example 6 - Interval Coalesce:

```
Base tables: inter(S: int, E: int)
CREATE VIEW lstart(T) AS
  (SELECT a.S FROM inter a, inter b
   WHERE a.S <= b.E
   GROUP BY a.S HAVING a.S = min(b.S))

WITH recursive coal (S, max() AS E) AS
  (SELECT lstart.T, inter.E FROM lstart, inter
   WHERE lstart.T = inter.S)
   UNION
  (SELECT coal.S, inter.E FROM coal, inter
   WHERE coal.S <= inter.S AND inter.S <= coal.E)
SELECT S, E FROM coal
```

The Interval Coalesce query finds the smallest set of intervals that cover input intervals. It is one of the most frequently used queries in temporal databases. However, it is notoriously difficult to write correctly in SQL [ZWZ06]. Here, we express it succinctly with RaSQL using the *max* aggregate.

In the first part, a non-recursive view called *lstart* is created to find all left start points of intervals that are not covered by other intervals (except itself), using a self-join on the base relation *inter*. In the second part, the recursive view *coal*, which represents the final coalesced intervals, is computed by extending the end points of intervals having the left starting points in *lstart* through iteratively merging other intervals which cover these end points.

Example 7 - Party Attendance:

```
Base tables: organizer(OrgName: str)
```

```
            friend(Pname: str, Fname: str)
WITH recursive attend(Person) AS
  (SELECT OrgName FROM organizer)
   UNION
  (SELECT Name, Ncount FROM cntfriends
   WHERE Ncount >= 3),
recursive cntfriends(Name, count() AS Ncount) AS
  (SELECT friend.FName, friend.Pname
   FROM attend, friend
   WHERE attend.Person=friend.Pname)
SELECT Person FROM attend
```

The Party Attendance query computes people who will attend a party — a person will attend the party if and only if three or more of his/her friends attend, or he/she is the organizer. This query is more complex than previous examples as it is expressed in a *mutual recursion* fashion, i.e., the definitions of two or more recursive relations reference to each other. The *attend* relation records people who will attend the party, which is computed from the information provided by *cntfriends*. The *cntfriends* relation records the number of a person's friends who will attend the party, which also needs the information from the *attend* relation.

Example 8 - Company Control:

```
Base tables: shares(By: str, Of: str, Percent: int)
WITH recursive cshares(ByCom, OfCom, sum() AS Tot) AS
  (SELECT By, Of, Percent FROM shares)
   UNION
  (SELECT control.Com1, cshares.OfCom, cshares.Tot
   FROM control, cshares
   WHERE control.Com2 = cshares.ByCom),
```

```
recursive control(Com1, Com2) AS
  (SELECT ByCom, OfCom
   FROM cshares
   WHERE Tot > 50)
SELECT ByCom, OfCom, Tot FROM cshares
```

The Company Control query was proposed by Mumick, Pirahesh and Ramakrishnan [MPR90] to calculate the complex controlling relationships between companies. Companies can purchase shares of other companies. In addition to the shares that a company owns directly, a company A owns shares that are controlled by a company B if A has a majority (over 50% of the total number) of B's shares. A tuple (A, B, 51) in the base relation *shares* indicates company A directly owns 51% of the shares of company B. This query also uses *mutual recursion*: The *cshares* view recursively computes the percentage of shares that one company owns of another company while the *control* view decides whether one company controls another company.

Example 9 - Same-Generation (SG):

```
Base tables: rel(parent: int, child: int)
WITH recursive sg (X, Y) AS
  (SELECT a.Child, b.Child FROM rel a, rel b
   WHERE a.Parent = b.Parent AND a.Child <> b.Child)
   UNION
  (SELECT a.Child, b.Child FROM rel a, sg, rel b
   WHERE a.Parent = sg.X AND b.Parent = sg.Y)
SELECT X, Y FROM sg
```

The Same-Generation (SG) query identifies pairs of nodes where both are the same number of hops from a common ancestor. The base relation *rel* includes the parent-child relationship of two nodes. In the base case, the recursive view *sg* is initialized with nodes which share the same parent; In the recursive case, the *sg* view is iteratively populated by new pairs of

nodes whose parents have already been classified as the same generation.

Example 10 - Reachability (Reach):

```
Base tables: edge(Src: int, Dst: int)
WITH recursive reach (Dst) AS
  (SELECT 1)
   UNION
  (SELECT edge.Dst FROM reach, edge
   WHERE reach.Dst = edge.Src)
SELECT Dst FROM reach
```

The Reachability (Reach) query basically does a Breadth-First Search (BFS) to find out all nodes that are reachable from the source node. In the base case, it adds the source node, i.e. 1, to the *reach* relation. In the recursive case, new nodes are added to the *reach* relation in each iteration if their neighbors have already been included in the relation, i.e., been visited during the BFS process.

Example 11 - All-Pairs Shortest-Path (APSP):

```
Base tables: edge(Src: int, Dst: int, Cost: double)
WITH recursive path (Src, Dst, min() AS Cost) AS
  (SELECT Src, Dst, Cost FROM edge)
   UNION
  (SELECT path.Src, edge.Dst, path.Cost + edge.Cost
   FROM path, edge
   WHERE path.Dst = edge.Src)
SELECT Src, Dst, Cost FROM path
```

The All-Pairs Shortest-Path (APSP) query describes the Floyd-Warshall's [flo] algorithm to compute the shortest paths between all pairs of nodes in a weighted graph. In the base case, the recursive relation `path` is initialized by all existing edges with costs. In the recursive

case, new paths are iteratively derived by joining already computed paths with edges, which are connected by the same intermediate node, with the costs being summed up. The *min* aggregate is used to find the minimal path among all paths between two nodes.

3.5 Compiler Implementation

The compilation of a non-recursive SQL query typically goes through three steps: (1) Parsing the query into an Abstract Syntax Tree (AST); (2) Analyzing the logical plan to resolve column and table references for each operator; (3) Performing physical planning to generate the execution plan. Though the general procedure for compiling a RaSQL query is similar, our recursive extensions bring several new challenges.

Challenges The SQL logical analyzer is unable to recognize a recursive view definition, i.e. a *self-reference* in the CTE will lead to infinite loops during *reference resolution*. Moreover, the iterative nature of the query plan generated from the *recursive case* requires to be executed multiple times until a fixpoint is reached, as described in Algorithm 3.

To overcome these challenges, we design a special analysis step called *Recursive Clique Analysis* for recursive CTEs, which makes the analyzer aware of the recursive definition to avoid resolution issues. In order to evaluate the *recursive plan* iteratively, we introduce a new *fixpoint operator* which drives the iterative evaluation process. It makes the output stage of an iteration serve as the input stages of the next iteration to ensure the correct dataflow between iterations. Once the fixpoint is reached, the operator returns the union of tuples produced during all iterations as the result.

We implement the RaSQL compiler and optimizer on top of Spark SQL[6]. We choose Apache Spark as the RaSQL compiler and runtime mainly because it supports the ANSI-SQL standard with a well-designed SQL parser, analyzer and optimizer. The compiled form of a

[6]Details about Apache Spark and Spark SQL is provided in Section 2.3

Spark SQL query is an RDD DAG which is targeted for distributed execution. However, we would like to point out that most of the compilation and optimization techniques proposed for RaSQL are also applicable to other RDBMS systems such as MySQL or Postgres.

3.5.1 Recursive Clique Analysis

Each recursive view defined in the CTE is transformed into a *Recursive Clique Plan* before any other analysis steps. In detail, it includes three main tasks: (1) Resolving recursive relation references; (2) Pushing down (pre-mapping) any aggregates used in recursive view columns to each sub-query; (3) Distinguishing between the *base* and *recursive* cases.

This transformation step is essential because: (1) The Spark SQL analyzer is unable to handle recursive relations, though as long as the recursive relations are resolved, most of the subsequent analysis and optimization steps can proceed normally; (2) An efficient execution of recursive queries with aggregates requires a *pre-mapping* of aggregates (PreM property in Section 3.3) into each recursive iteration. Thus, the corresponding group-by keys and aggregate columns need to be assigned for each sub-query; (3) The physical planner and the execution engine take different strategies to compile and execute *base* and *recursive* cases, thus an early separation of the two cases will greatly ease subsequent steps.

We show the *Recursive Clique Plan* (Figure 3.2 (a)) generated by the recursive clique analysis of the `waitfor` query ($Q2$). The entire query is transformed into an operator tree with the *recursive clique* operator as the root. The sub-query which only refers to the base relation `basic` is recognized as the *base case*, while the other sub-query with the *recursive-reference* is recognized as the *recursive case*. Note that the *max* aggregate is pushed down into each sub-query with the group-key and aggregation column correctly mapped to `Part` and `Days`. Other operators such as *join* are directly translated from their counterparts in the original query.

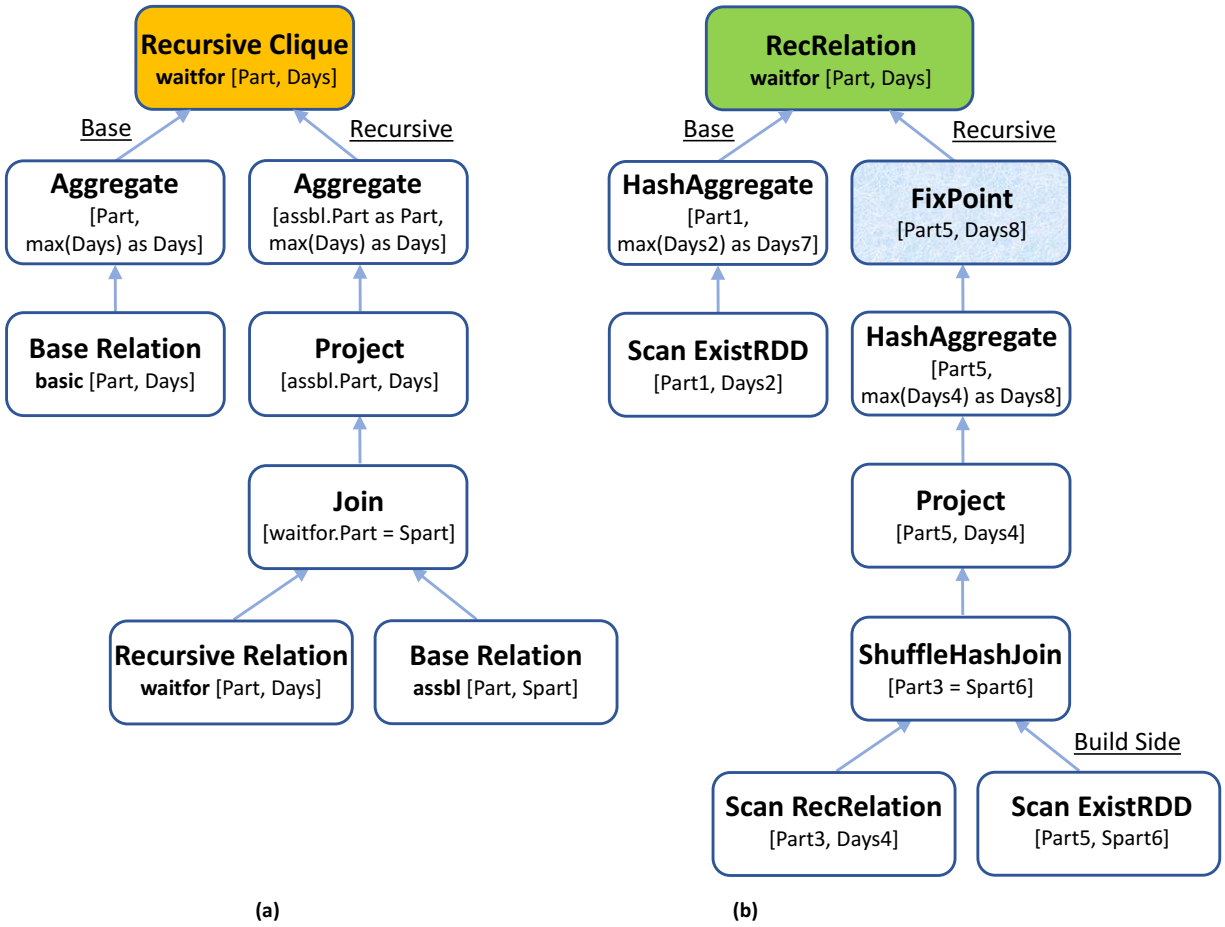


Figure 3.2: RaSQL Query Plan - (a) Clique (b) Physical

3.5.2 Physical Plan Generation

The *Recursive Clique Plan* is further analyzed by other rules defined in the analyzer, such as resolving column aliases, converting basic operators (e.g. join, filter, aggregate) to their logical operator counterparts, etc. The analyzed plan is then optimized by a batch of rules such as predicate pushdown, filter combination and constant evaluation defined in the optimizer. Finally, the logical plan is sent to the physical planner to generate the final execution plan.

Compared to the logical plan, the most noteworthy change is the inclusion of the *fixpoint* operator in the physical plan (Figure 3.2 (b)). The *fixpoint* operator is the center of the recursive iteration: it evaluates its child plan (the *recursive case*) repetitively until the fixpoint is reached. The execution of each iteration takes place *eagerly* as the *fixpoint* operator

needs to know if the fixpoint is reached to decide whether to schedule the next iteration or not. The distributed evaluation of the *fixpoint* operator and its optimization is discussed in Section 3.6. Other important decisions made in the physical planning phase include the join type selection, e.g., HashJoin vs. SortMergeJoin (Section 3.6.3) and the insertion of *exchange* operators [Gra94]. Due to space limitations in the figure, we omit showing all *exchange* operators. The main function of the *exchange* operator is to shuffle the output of a child operator to the desired partitioning as required by its parent operator for parallel execution, which is discussed in Section 3.6.

3.6 Fixpoint Operator Execution

The naive evaluation described in Algorithm 3 provides a conceptual idea for evaluating a recursive query. However, due to its inefficiency, the *fixpoint operator* adopts an optimized version called Semi-Naïve evaluation (SN) [Ban86].

The main idea of SN is the *delta evaluation*: only newly produced data items stored in the *delta* relation at iteration i will participate in the calculation for iteration $i+1$, thus much less computation is required compared to the naive version given both use the same number of iterations to the fixpoint.

We first revisit the classical single-node Semi-Naïve evaluation method using the Transitive Closure (TC) example (Algorithm 4). The optimization made to the distributed execution is discussed in Section 3.6.1 and the optimization of aggregates in recursion is described in Section 3.6.2, the choice of distributed join types is discussed in Section 3.6.3.

```
Base tables: edge(Src, Dst)
WITH recursive tc (Src, Dst) AS
  (SELECT Src, Dst FROM edge) UNION
  (SELECT tc.Src, edge.Dst FROM tc, edge
   WHERE tc.Dst = edge.Src)
```

```
SELECT Src, Dst FROM tc
```

Transitive Closure (TC) Query Example

Algorithm 4 Semi-Naive Evaluation of TC

1: $\delta\texttt{tc} \leftarrow \texttt{edge}(\texttt{X}, \texttt{Y})$

2: $\texttt{tc} \leftarrow \delta\texttt{tc}$

3: **DoWhile**

4: $\quad \texttt{tc}_{\texttt{new}} \leftarrow \pi_{\texttt{X,Y}}(\delta\texttt{tc}(\texttt{X}, \texttt{Z}) \bowtie \texttt{edge}(\texttt{Z}, \texttt{Y}))$

5: $\quad \delta\texttt{tc}' \leftarrow \texttt{tc}_{\texttt{new}} - \texttt{tc}$

6: $\quad \texttt{tc} \leftarrow \texttt{tc} \cup \delta\texttt{tc}'$

7: $\quad \delta\texttt{tc} \leftarrow \delta\texttt{tc}'$

8: **EndDoWhile**$(\delta\texttt{tc} \neq \varnothing)$

9: **return** tc

The Transitive Closure query generates all pairs of nodes $(\texttt{X}, \texttt{Y})$ that are reachable from X to Y. In Algorithm 4, tc is the set of all tuples produced for the recursive relation and $\delta\texttt{tc}$ ($\delta\texttt{tc}'$) is the set of new tuples (delta) produced for the current iteration. In SN, the *base case* is evaluated first. The tuples in edge become the initial set of tuples for both $\delta\texttt{tc}$ and tc (line 1-2). Then, SN iterates until a fixpoint is reached (line 8). Each iteration begins by joining $\delta\texttt{tc}$ with the *base* relation relation edge and projecting $\texttt{X}, \texttt{Y}$ terms to produce candidate tc results (line 4). These results are then set-differenced with tc to eliminate duplicates and produce $\delta\texttt{tc}'$ (line 5), which is unioned into tc (line 6) and becomes $\delta\texttt{tc}$ (line 7) for the next iteration. As each iteration only uses the $\delta\texttt{tc}$ as the join input in SN, no duplicate tuples are generated, thus the intermediate results are much smaller than the naive version, leading to a more efficient execution. It is worth mentioning that the SN evaluation divides the recursive relation into a *delta* relation ($\delta\texttt{tc}$) and an *all* relation (tc) during evaluation, which are two important notions and will be used frequently in later discussions.

Algorithm 5 Distributed Semi-Naïve (DSN) Evaluation

B: Base relation
R: Recursive relation (All)
δR, $\delta R'$: Recursive relation ($Delta$)
K: Partition key for δR, $\delta R'$, B, R, also the join key

function MapStage(δR, B)
    ▷ Require: δR, B co-partitioned on join key K
    for each partition pair of $(\delta R, B)$ **do**
        emit $\Pi(\delta R \bowtie_{\delta R.K=B.K} B)$

function ReduceStage($\delta R'$, R)
    ▷ Require: $\delta R'$, R co-partitioned on key K
    for each partition pair of $(\delta R', R)$ **do**
        $D \leftarrow \delta R' - R$
        $R \leftarrow \delta R' \cup R$
        emit D

$\delta R \leftarrow$ Results of Base Case, $R \leftarrow \emptyset$
do
    $i \leftarrow i + 1$
    $MapOutput \leftarrow$ MapStage(δR, B)
    $\delta R' \leftarrow$ ShuffleExchange($MapOutput$, key $= K$)
    $\delta R \leftarrow$ ReduceStage($\delta R'$, R)
while $(\delta R \neq \emptyset)$
return R

3.6.1 Distributed Semi-Naive Evaluation

To derive a distributed version of the Semi-Naive evaluation (Algorithm 4), it is necessary to identify the key operations that the algorithm does in each iteration, which can be generalized to two steps: (1) the *delta* relation produced at iteration *i-1* is joined with the *base* relation (plus other RA operations such as filter, project if necessary); (2) the result is set differenced with the *all* relation to produce the new *delta* relation for iteration *i+1* and set union-ed to expand the *all* relation. Besides, the data exchange between iterations also need to be meticulously planned in order to achieve the optimal performance. Thus we discuss our efforts in improving the intra/inter-iteration planning beyond the Spark's default execution model.

Intra-Iteration Planning The evaluation steps within an iteration can be naturally modeled as a Map stage and a Reduce stage, and supported on any distributed system that adopts the idea of MapReduce [DG08] (e.g. Hadoop, Spark), which leads to the DSN evaluation as shown in Algorithm 5: In the Map stage, the join and other RA operations will generate the Map outputs (line 9), which are then *shuffled* to the desired partitioning (formal definition of partitioning the relational dataset is provided in Section 2.3) as required by the reducers (line 22); In the Reduce stage, input tuples are set-differenced/unioned with the existing tuples in the *all* relation to produce the new *delta* relation and *all* relation (line 14, 15) for the next iteration. All computations are performed partition-wise, and the join and set operations require both input relations co-partitioned (line 7, 12).

Inter-Iteration Planning In principle, as long as the output result of the previous iteration to be served as the input of the next iteration, the scheduler will take care of the inter-iteration planning. However, extra challenges are posed in practice which significantly affect the performance: Firstly, the *Spark scheduler is unaware of the nature of fixpoint iteration jobs*, which leads to the stages of each iteration to be scheduled independently without considering the inter-iteration data locality. Secondly, as a Spark RDD is immutable, each union or set difference operation will result in a new RDD being created with most of its data re-

dundantly copied from up-stream RDDs. These two issues nearly eliminate the performance benefits brought by the Semi-Naive evaluation, thus we designed a special data structure for RDDs and a new scheduling policy to optimize the Inter-Iteration Planning.

SetRDD We designed a new data structure for the *all* RDD, which organizes each of its partition data in an append-only hashSet to support frequent set-difference/union operations. Each partition of the *all* RDD is cached in workers' memory or disk during all iterations to enable fast access. As a result, the speed of set operations is greatly improved because the set union only incurs the overhead of adding new data items. This is much faster than having to copy all data as required by immutable RDDs.

A natural concern that arises is whether the fault-recovery of the RDD is compromised by the fact that the lineage of the *all* RDD is no longer preserved due to its mutable structure. In reality, a good recovery speed can still be achieved since its set data which represents the computed results from all previous iterations are always cached (checkpointed), a failure in any iteration will only incur the replay of the execution job belonging to the current stage, resulting in a competitive performance w.r.t. lineage-based method.

Partition-Aware Scheduling The Algorithm 5 poses a strong partitioning restriction for the *base* relation and *delta* relation, which demands the reduce key to be the same as the join key. By doing this, the reduce output will be partitioned as required to be served as input for the next iteration's Map tasks, which means that an ideal scheduling policy could take advantage of this to schedule a Map task of the next iteration to the worker which contains its input data, achieving the *iter-iteration data locality*.

However, the Spark scheduler fails to do so because by default a hybrid strategy is adopted to decide where to run a task by considering a combination of factors such as the workload of executor, the locality waiting time, and the input data locations. Though this strategy generally works well in scheduling multiple independent jobs, it leads to sub-optimal performance in fixpoint iterations due to the ignorance of the *iter-iteration data locality* which causes unnecessary remote data fetches across iterations.

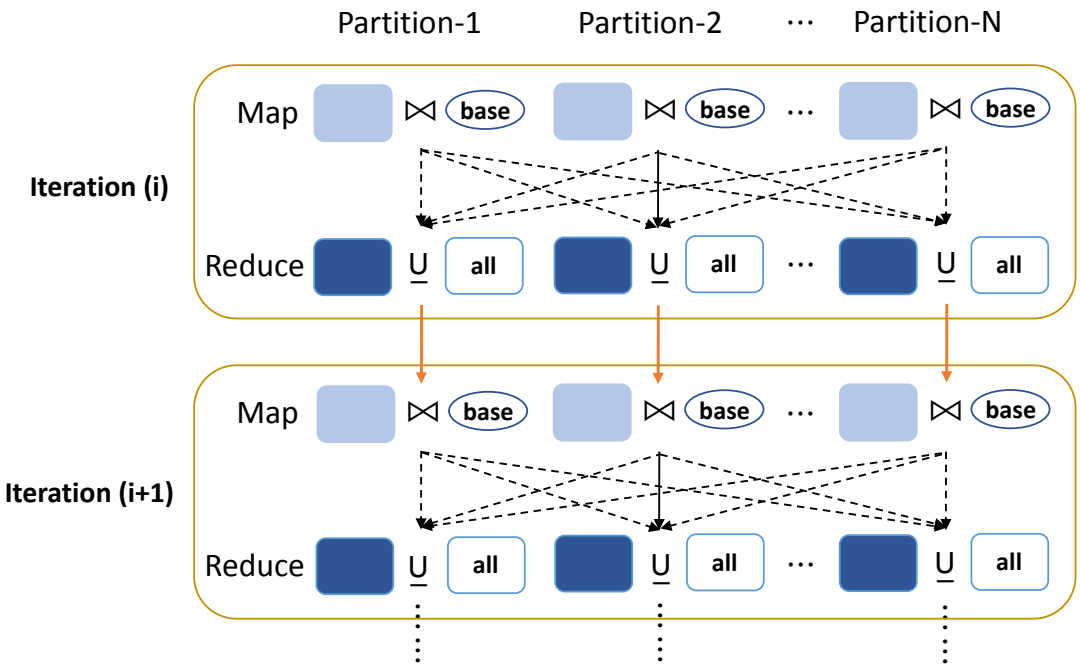


Figure 3.3: Dataflow of DSN iterations

Thus we designed a new scheduling policy, which makes the Spark scheduler aware of the locations of a *cached* RDD's partition blocks, and schedules corresponding tasks of another RDD to these locations. It works as follows: 1) when an RDD is cached, it sends back the locations of its partitions to the master; 2) if another RDD needs to be *co-partitioned* with this RDD in case of join, union, etc, it can request the scheduler to assign its tasks to the *cached* RDD's locations.

Figure 3.3 shows the dataflow between two consecutive iterations i and $i+1$. The *base* relation (ellipse) is partitioned and cached on each worker before the start of the first iteration, and it is joined with the *delta* relation (blue rectangle) to produce the new *delta* relation in each Map stage. The *all* relation (unfilled rectangle) is partitioned and cached using SetRDD, and is expanded in each Reduce stage through the union with the new *delta* relation. Thanks to the *partition-aware scheduling* policy, the Map task which reads the partitioned data produced from the Reduce task of the previous iteration are always scheduled on the same worker, which eliminates any unnecessary remote fetches and achieves the data-locality across iterations.

Algorithm 6 Map/Reduce Stage w/ Max Aggregate

1: **function** MAPSTAGE(δR, B)
2: ▷ Require: δR, B co-partitioned on join key K
3: **for each** partition pair of $(\delta R, B)$ **do**
4: $P \leftarrow \Pi_{k,v}(\delta R \bowtie_{\delta R.K=B.K} B)$
5: **emit** *Partial_Aggregate*(P, func=max, key=k)
6:
7: **function** REDUCESTAGE($\delta R'$, R)
8: ▷ Require: $\delta R'$, R co-partitioned on key K
9: **for each** partition pair of $(\delta R', R)$ **do**
10: **for each** partial aggregated (k, v) of $\delta R'$ **do**
11: **if** k not in $R.keys$ **then**
12: put (k, v) in R, add to D
13: **else if** $v > R(k)$ **then**
14: update (k, v) in R, add to D
15: **emit** D

3.6.2 Aggregates In Recursion

To efficiently support the aggregates-in-recursion optimization enabled by the PreM property, the general logic of Map and Reduce tasks need to be extended as shown in Algorithm 6: In the Map stage, the projected tuples will be *partially aggregated* first to reduce the shuffling data size (line 5); In the Reduce stage, the *partially aggregated* tuples are *merged* with the existing results sharing the same aggregation key in *all* RDD (R). The *delta* RDD (D) includes tuples not only with newly generated group key (line 11-12), but also existing groups whose aggregate values are updated in the current iteration (line 13-14). These are indeed the extended operations for set difference and union under aggregates.

Again, we use the `waitfor` example to illustrate how delta tuples are generated if *max* ag-

gregates are used: Suppose the tuple $(b, 5)$[7] has already been produced in previous iterations, which indicates the maximum waitfor days calculated for part "b" is 5 so far. If a bigger waitfor day is derived from its subpart in the current iteration, e.g., 6, then $(b, 6)$ will be added to δ, and participates in the next iteration's calculation. However, if a produced waitfor day is smaller, e.g., 3, then $(b, 3)$ will be ignored and discarded due to the property of monotonic aggregates. For theoretical proof of the correct semantics for semi-naive evaluation with monotonic aggregates in recursion, we refer [MSZ13] for interested readers.

3.6.3 Join Implementation

The join operation between the *delta* RDD and *base* RDD in the Map stage (line 4 in Algorithm 6) is typically the most time consuming part in each iteration. Here we consider two types of distributed joins — shuffle-hash join and sort-merge join with their performances compared. An optimized version of the broadcast-hash join is a better choice for certain types of queries, which is discussed in detail in Section 3.7.

Shuffle-Hash Join The general idea of the shuffle-hash join is that one side (build side) of the join will build a hash table, and the other side (stream side) streams the tuples and probes the possible matches in hashed relation. In our implementation, the *base* relation side is always chosen as the build side due to two reasons. First, the size of the *delta* relation will become very large during the recursion iterations. Though its size will reduce to zero when the fixpoint is reached, it is still much larger than the *base* relation in most iterations. Second, the fixed build side allows the hash table to be only created once and then cached/reused across iterations, thus the actual build time is amortized to small when the number of iterations accumulates up.

Sort-Merge Join If both join sides contain very large relations that cannot fit in memory, sort-merge join becomes a better choice over shuffle-hash join as it uses less memory and has better cache locality. The join process starts with sorting both inputs, followed by a merge

[7] The grouping key is "b" and aggregate value is 5.

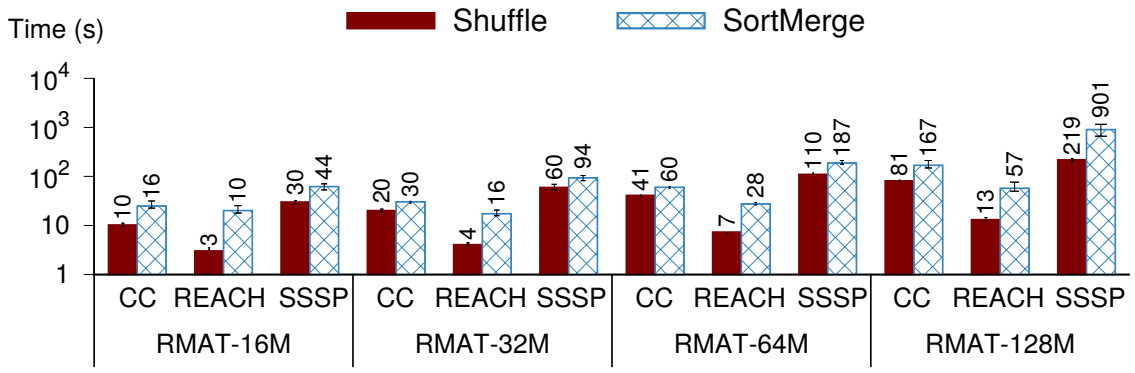


Figure 3.4: Shuffle-Hash Join vs. Sort-Merge Join

phase of the sorted runs. In our implementation, the *base* relation side is also *cached*, i.e. sorted once and reused during iterations.

We did performance comparisons between the two joins using three queries (CC, SSSP, REACH) on differently sized `RMAT` dataset[8]. The performance in Figure 3.4 shows shuffle-hash join always performs better. This result is not surprising as we always allocate enough memory for the base relation side to build the entire hash map, which shows the cost of probing the matches in the hash table is generally smaller than sorting the whole delta relation. We also observed that the sort-merge join requires much less memory than the shuffle-hash join, and more stable than the latter, i.e. causes less job failures, indicating it is a better choice if the join size is large or stability is preferred.

3.7 Performance Optimization

After our initial experiments, we found that the major performance affecting factors not only include the *computation-intensive* operations such as joins and aggregates, but also *IO-intensive* tasks such as scheduling and shuffling. Thus we designed hybrid optimization strategies such as stage combination, decomposed plan optimization and code generation, to alleviate the performance bottlenecks in both aspects.

[8]Experiment settings are provided in Section 3.8.

Algorithm 7 Optimized DSN Evaluation w/ Aggregate

B: Base relation
R: Recursive relation (All)
δR: Recursive relation ($Delta$)
K: Partition key for δR, B, R, also the join key

function ShuffleMapStage(δR, B, R)
    ▷ Require: δR, B, R all-partitioned on key K
    for each partition pair of $(\delta R, B, R)$ **do**
        for each partial aggregated (k, v) of δR **do**
            if k not in $R.keys$ **then**
                put (k, v) in R, add to D
            else if $v > R(k)$ **then**
                update (k, v) in R, add to D
        $P \leftarrow \Pi_{k,v}(D \bowtie_{D.K=B.K} B)$
        emit *Partial_Aggregate*(P, func=max, key=k)

$\delta R \leftarrow$ Results of Base Case, $R \leftarrow \emptyset$
do
    $i \leftarrow i + 1$
    $MapOutput \leftarrow$ ShuffleMapStage(δR, B, R)
    $\delta R \leftarrow$ ShuffleExchange($MapOutput$, key $= K$)
while ($\delta R \neq \emptyset$)
return R

3.7.1 Stage Combination

The abstraction of the distributed Semi-Naive evaluation process as a series of Map/Reduce stages (Algorithm 5) provides a general implementation guidance on any systems that support the MapReduce model. In Spark, however, the RDD model is more powerful than

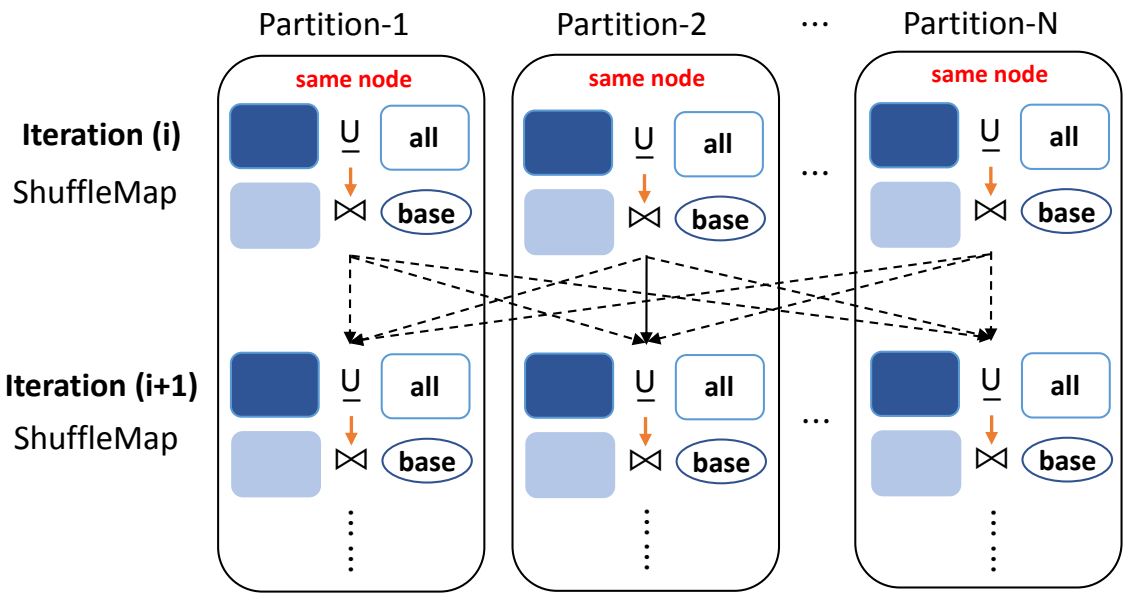


Figure 3.5: Dataflow of optimized DSN iterations

MapReduce as all RDD transformations that are not separated by the *shuffling* operation can be pipelined within a stage. Thus, we can combine the Reduce stage of the iteration i with the Map stage of the iteration $i+1$ into a single ShuffleMap stage as shown in Algorithm 7. Note that *stage combination* is only possible by activating the *partition-aware scheduling* policy introduced in Section 3.6.1 because all three RDDs δR, B, R that participate in the evaluation need to be co-partitioned and any specific partition requires to be scheduled on the same worker across all iterations to minimize the data movements.

The optimized distributed Semi-Naive evaluation is visualized in Figure 3.5. Note that each iteration now only takes a single stage thus greatly reduces the overall scheduling cost and improves the cache locality. We evaluate the effect of stage combination optimization using CC, REACH and SSSP queries on various sizes of `RMAT` dataset (Figure 3.6). The result shows a very significant improvement on the overall execution time. For recursive queries without aggregates, such as REACH, it achieves a performance boost of 3X to 5X. For queries with aggregates such as CC and SSSP, it achieves an improvement between 1.5X to 2X.

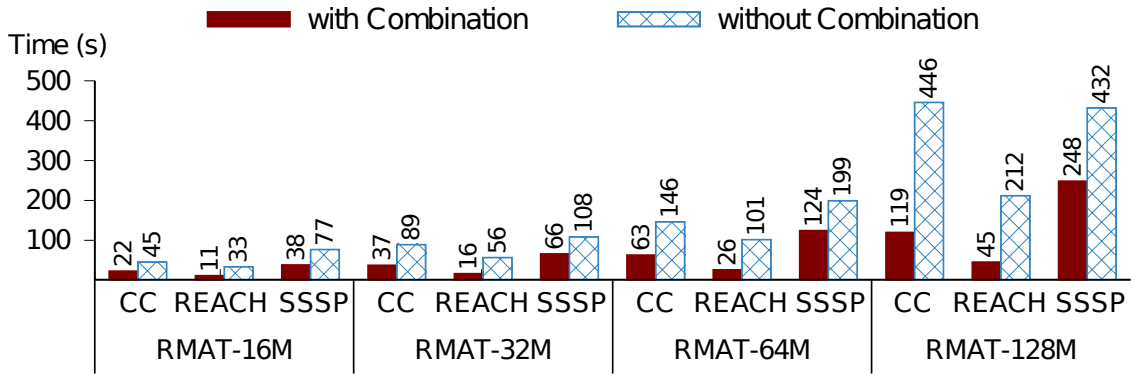


Figure 3.6: Effect of Stage Combination

3.7.2 Decomposed Plan Optimization

Certain kinds of recursive queries can undergo special optimizations as indicated by previous research work [46][57]. These queries can be compiled into *decomposable* plans which exhibit an attractive feature for parallel execution: A well-chosen partitioning strategy for the *decomposable* plan allows the result RDD produced from the join to preserve the original partitioning of the input *delta* RDD. For example, the plan of the linear TC query is *decomposable*, thus if δtc(X, Y) is partitioned on X, δtc(X, Y) can be joined with *base* relation edge(Y, Z) to produce the result δ'tc(X, Z), with the same partitioning as the input δtc(X, Y) (both partitioned on X).

As the output *delta* RDD preserves the input's partitioning, the executor which works on partition i in the current iteration can continue to work on the same partition in the next iteration, with all its input data fetched locally. This is a pretty nice feature as it allows all partitions of a *decomposable* plan to be computed out independently without the global synchronization, i.e. each executor can claim a partition and performs the iterative computation on its own without communication with the master or other workers until the fixpoint is reached.

However, a *decomposable* plan often implies the join key is different from the partitioning key. For example, the δtc(X, Y) is partitioned on X but joined on Y, which indicates that it is possible to join with any tuple of the *base* relation edge(Y, Z). To fulfill this requirement, each worker must own an entire copy of the *base* relation, instead of just a partition of

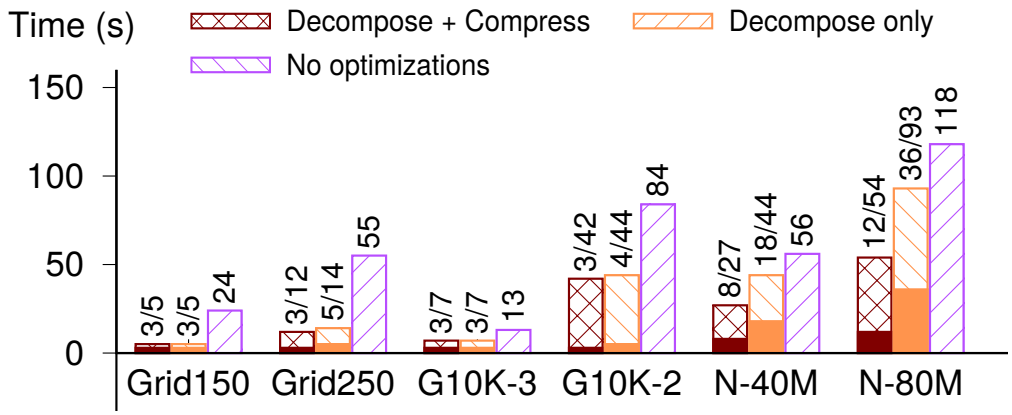


Figure 3.7: Effect of Decomposition and Compression

it, in order to conduct the independent execution. In practice, an entire copy of the *base* relation is distributed to each worker (and cached on it) before the start of the recursive iterations.

In our implementation, we use broadcast-hash join to distribute the *base* relation to workers and perform the hash join in each iteration. The default implementation provided by Spark requires the hash table to be built on the master node before being sent to workers, which is inefficient for large relations as the hashed relation is often 2X to 3X larger than the original one. We optimize the process by broadcasting the compressed relation and ask each worker to build the hash table on its own, thus minimizing the data transfered.

Figure 3.7 shows the effect of the *decomposed* execution and the broadcast compression by measuring the performance of `TC` query on various sizes of synthetic graphs (parameters are provided in Section 3.8). The broadcasting time is depicted on the graph as solid bars (broadcast/total time). The figure shows that even broadcasting a large relation takes time, the overall performance of the *decomposable* optimization is still much improved over the un-optimized version, by approximately 1.5X to 2X. Moreover, the broadcast compression contributes a lot in bringing a good performance on large graphs such as N-40M and N-80M, with a reduction of nearly half of the overall execution time.

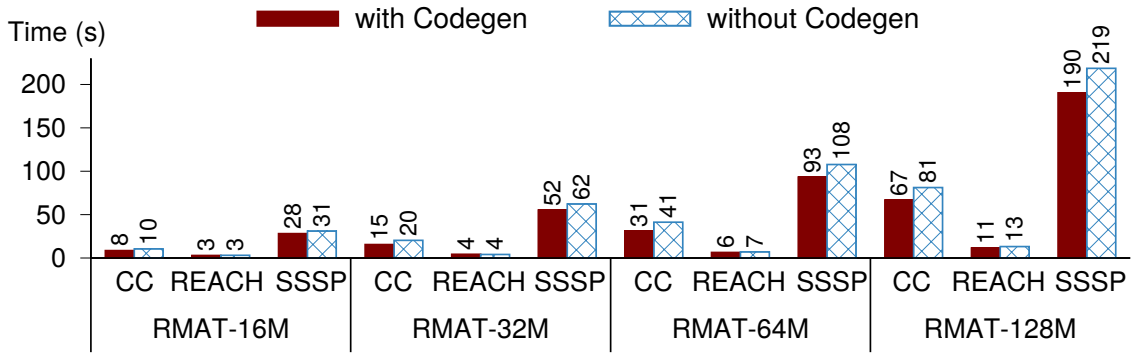


Figure 3.8: Effect of Code Generation

3.7.3 Code Generation

Spark 2.0 [ZCF10] introduces the whole-stage code generation which helps the execution engine eliminate the bottlenecks of the classical volcano iterator model [GM91] such as frequent virtual function calls, and better leverages CPU registers for intermediate data [Neu11]. The engine uses the code generated at the runtime to compute the query results instead of using the actual operators, while the code is generated from collapsing fragments of query operators into a single function wherever possible.

Since a RaSQL query is compiled to a Spark SQL plan for execution, queries written in RaSQL can also benefit from the speedups brought by the whole-stage code generation. Note that to fully leverage the power of code generation, we add extra code generation rules for operators without code generation support in Spark, such as shuffle-hash join with *base* relation cached. As a result, for most queries, the code generation is able to collapse all operators within each iteration into a single function to achieve the best performance.

To demonstrate the effect of code generation, we compare the pure recursive iteration time (excluding the data loading time) of CC, SSSP and REACH queries on `RMAT` datasets, as shown in Figure 3.8. For CC and SSSP queries, the code generation reduces the computation time by 10% to 20%. However, the effect of the code generation is not as significant as other optimizations, mainly due to two reasons. First, as code generation collapses multiple RDD transformations into a single function call, only large number of RDD transformations will make a distinguished effect, which is not our case. Second, most queries used in our experi-

ments are not computation-intensive, thus a large portion of the time is spent on IO-intensive tasks such as shuffling, which further diminishes the effectiveness of code generation.

3.8 Experiments

We evaluate the performance of RaSQL system by comparing the query execution time with five other systems or implementations, namely, Apache Giraph, GraphX, BigDatalog, Myria and Spark-SQL-SN on various sizes of synthetic and real world datasets.

Apache Giraph and GraphX are the two most widely used distributed graph processing engines. Both of them are inspired by Google's Pregel [MAB10] system, but built on top of different execution models, i.e. Hadoop MapReduce vs. Spark RDD. These two systems exhibit good performance in processing large-scale graph datasets as demonstrated by [GXD14, LCY14]. We choose distributed graph engines for comparison mainly because many graph algorithms can be elegantly expressed as recursive queries in RaSQL (Section 3.4).

We choose two systems from academia for comparison, BigDatalog [SYI16] and Myria [WBH15]. BigDatalog is a recursive Datalog query engine, and the implementation of RaSQL system borrows good ideas from BigDatalog's execution engine but with a new architecture and optimizations introduced in Section 3.7 — we want to show the improvement brought by these optimizations over BigDatalog. Myria is a distributed big data management system focused on complex workflow processing, which also supports recursive queries expressed by Datalog. For baseline comparison, we write optimized Spark SQL programs to simulate the semi-naive evaluation using a mix of the Scala loops and SQLs, namely Spark-SQL-SN, as Spark SQL does not support recursive queries directly.

For other non-distributed graph engines such as Neo4j, and RDBMS systems such as MySQL, our initial experiments on them present a far worse performance (at least an order of magnitude slower) compared with other systems due to their poor parallel support for recursive

queries. Since they cannot run in distributed mode[9], we are unable to tune them to run fast enough to perform any meaningful comparisons with other systems on our test queries and datasets.

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

For each system, one node is dedicated as the master and other 15 nodes as the workers. Each worker node is allocated 30 GB RAM and 8 CPU cores (120 total cores) for execution. Myria is configured with one instance of Myria and PostgreSQL per node since each node has one disk. The Hadoop version is 2.2 for all systems using HDFS. We evaluate RaSQL, BigDatalog, GraphX and Spark-SQL programs with one partition per available CPU core, on Spark 2.0 platform. The Giraph system directly runs as MapReduce jobs on Hadoop. All systems are activated with the in-memory computation by default. RaSQL is configured to execute queries using shuffle-hash join and optimized DSN evaluation with stage combination and code generation optimizations.

Dataset Parameters. We use various sizes of synthetic datasets to verify the effect of different optimization approaches as listed in Table 3.1. `Tree11` contains trees of height 11, and the degree of a non-leaf vertex is a random number between 2 and 6. `Grid150` is a 151 by 151 grid while `Grid250` is a 251 by 251 grid. The Gn-e graphs are n-vertex random graphs (Erdős-Rényi model) generated by randomly connecting vertices so that each pair is connected with probability 10^{-e}. Note that although these graphs appear small in terms of number of vertices and edges, TC and SG are capable of producing result sets many orders of magnitude larger than the input dataset, as shown in the last two columns.

As it is generally difficult to tune each system to the optimal performance, we have tried our

[9]In fact, Neo4j can utilize Apache Spark as its distributed execution engine [cyp]. However, the actual computation tasks are executed by the libary code written in Spark SQL or GraphX programs (which are already included in our experiments) with the input data fetched from the Neo4j storage.

Name	Vertices	Edges	TC	SG
Tree11	71,391	71,390	805,001	2,086,271,974
Grid150	22,801	45,300	131,675,775	2,295,050
Grid250	63,001	125,500	1,000,140,875	10,541,750
G10K-3	10,000	100,185	100,000,000	100,000,000
G10K-2	10,000	999,720	100,000,000	100,000,000
G20K-3	20,000	399,810	400,000,000	400,000,000
G40K-3	40,000	1,598,714	1,600,000,000	1,600,000,000
G80K-3	80,000	6,399,376	6,400,000,000	6,400,000,000

Table 3.1: Parameters of Synthetic Graphs

best to minimize this effect. Nevertheless, we still emphasize that the purpose of our experiments is *not* to show RaSQL system is fundamentally faster than some other system, but to demonstrate that a general recursive query engine can be optimized to achieve competitive performance of special-purpose graph systems.

3.8.1 Graph Data Analytics

In this section, we show the performance comparison of our system with GraphX, Giraph, BigDatalog and Myria using three common graph analytics queries. We report the execution time of these systems over the benchmark programs on different sizes of synthetic and real world graphs.

Programs. We choose three commonly used graph queries — REACH, CC and SSSP (provided in Section 3.4) for performance comparison, mainly because they are commonly used, already implemented as library algorithms in most graph systems, have well-understood behavior to be served as fair measurements. The REACH program uses breadth-first-search to find all nodes that are reachable from a source vertex; the CC program uses label propagation algorithm to identify the connected components in the graph; the SSSP program computes

the shortest paths from a source node to all other nodes in the graph.

To give an overview of the time complexity of each algorithm, let m be the number of edges, n the number of vertices, and d the diameter of the graph, the number of intermediate results produced in each iteration is $O(m)$, $O(dm)$ and $O(nm)$ for REACH, CC and SSSP. Thus given a dataset, REACH is expected to run fastest among all three while CC in the middle and SSSP is the slowest as both the computation and communication costs grow linearly with the sizes of results produced in each iteration.

Datasets. For synthetic graphs, we use `RMAT` graph generator [gtg] with parameters (a, b, c) = (0.45, 0.25, 0.15) to generate different sizes of `RMAT`-n graphs ($n \in$ {1M, 2M, 4M, 8M, 16M, 32M, 64M, 128M}). `RMAT`-n has n vertices and $10n$ directed edges with uniform integer weights ranging from $[0, 100)$. The experiments on different sizes of `RMAT`-n datasets help us understand how RaSQL and other systems scale with graphs of increasing size. For real world graphs, we use four frequently experimented datasets as listed in Table 3.2.

Name	Vertices	Edges	Source
`livejournal`	4,847,572	68,993,773	[liv]
`orkut`	3,072,441	117,185,083	[ork]
`arabic`	22,744,080	639,999,458	[ara]
`twitter`	41,652,231	1,468,365,182	[KLP10]

Table 3.2: Parameters of Real World Graphs

We record the total time of evaluation starting from the data loading until the evaluation completes, as most systems do not explicitly report their data loading time. For CC, each point on the figure represents the average evaluation time on the test graph over 5 runs. For REACH and SSSP, each point represents the average time over 5 randomly selected vertices, run 5 times each.

Figure 3.9, 3.10 and 3.11 show the experimental results for the `RMAT`-n graphs. In general, we noticed that for all three programs on our test graphs, RaSQL system runs either the fastest

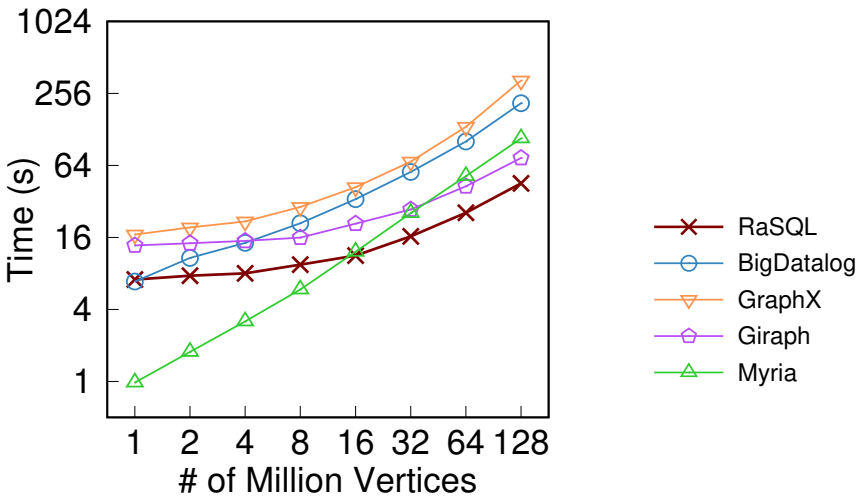


Figure 3.9: REACH Experiments on RMAT-n graphs

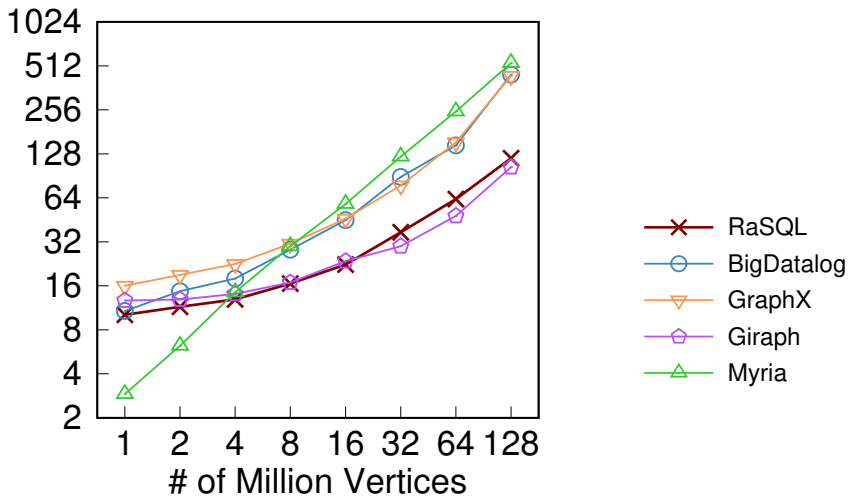


Figure 3.10: CC Experiments on RMAT-n graphs

(REACH) or very close (CC, SSSP), i.e. within 10% to the fastest system on RMAT-16M or larger sizes, outperforming GraphX by 4X to 8X; Giraph exhibits very similar performance curves with our systems on CC and SSSP queries, and always 1.5X to 2X slower than RaSQL on REACH; GraphX is generally slower than other systems but it still shows good scalability when the data size goes up; Myria runs fastest when the dataset size is small (typically 4M or smaller), but it scales poorly and gradually lags behind other systems on large data sizes. A potential explanation is that Myria has less overhead on small graphs but its communication module is less robust compared to other systems, thus affecting its performance on large datasets. In summary, RaSQL shares a similar curve with Giraph, which demonstrates

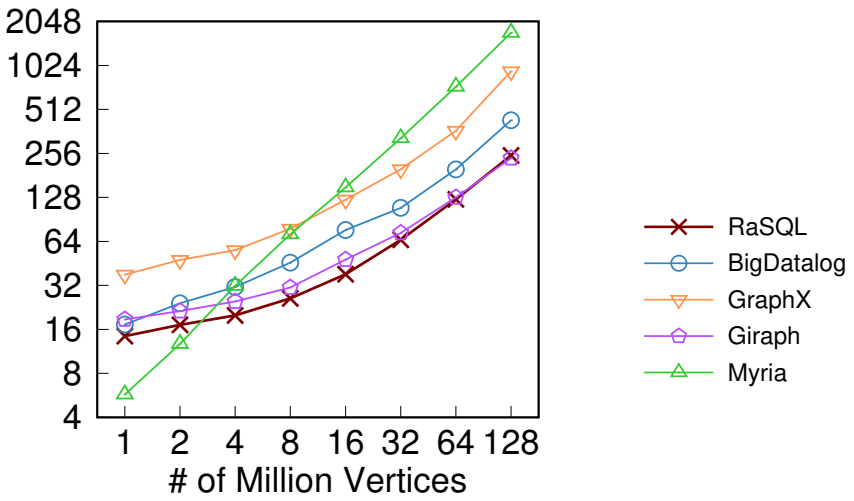


Figure 3.11: SSSP Experiments on RMAT-n graphs

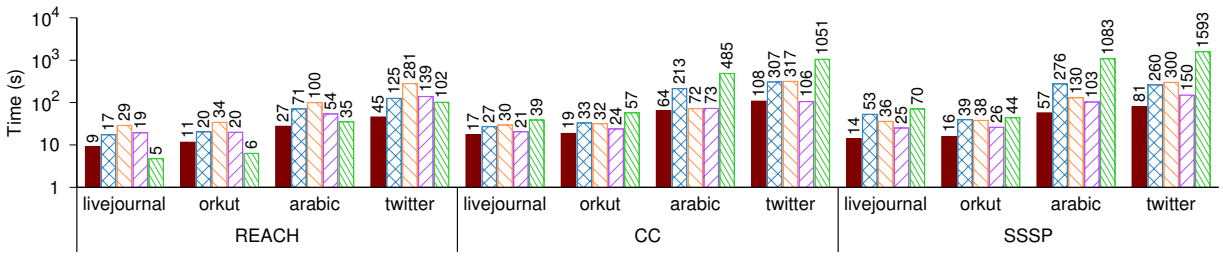


Figure 3.12: Systems performance comparison of REACH, CC and SSSP on real graphs.

the best scalability among all systems, and are always among the fastest systems on three programs when the data sizes become large.

Figure 3.12 shows the performance results for four real world datasets. Interestingly, RaSQL shows a better relative performance compared with Giraph: approximately 2X faster on REACH and SSSP. This improvement is largely in credit to a better handling of skewed datasets in RaSQL, i.e. a better partitioning strategy that leads to more balanced workloads on each executor. GraphX also closes its gap to Giraph, which is only 1.5X to 2X slower, but still much slower than RaSQL. Over all real world dataset experiments, RaSQL ranks the 1st on 9 tests, and ranks the 2nd on the remaining 3 tests, which demonstrates RaSQL's superior performance.

3.8.2 Complex Data Analytics

In this section, we report experimental results on three complex data analytics queries, to demonstrate the performance of RaSQL beyond the traditional graph applications.

Programs. We choose three real world application queries — Delivery, Management and MLM (The introduction example and Example 4, 5 in Section 3.4). The Delivery query describes the Bill-Of-Materials (BOM) scenario. The Management query computes the number of subordinates that a manager has in a company. The MLM calculates the bonus that a multi-level marketing company needs to pay for its members.

Datasets. As all three queries work on hierarchical data, we generate datasets of trees in different levels: each tree node has randomly 5 to 10 children, and each child has a 20% to 60% chance (*leaf probability*) of becoming a leaf. For Delivery query, the *basic* relation is generated by assigning weights to the leaf nodes and for MLM query, the *sales* relation is generated by assigning weights to each node in the tree. The final datasets we generated for experiments contain trees of height 10, 11, 12, 13 with 40M, 80M, 160M and 300M nodes.

As these three queries are not typical graph queries, we use GraphX, Spark-SQL-SN and Spark-SQL-Naive for the baseline comparisons because all of them execute on the same platform, i.e. Spark 2.0, as the RaSQL system. The Spark-SQL-SN/Naive programs are optimized Spark programs to simulate the semi-naive and naive recursive evaluation using a mix of the Scala loops and Spark SQLs.

Figure 3.13 shows the experiment results, not surprisingly, that RaSQL performs the best over all queries on all datasets, with at least 2X faster than GraphX on 40M and 80M datasets. The effectiveness of RaSQL optimizations becomes more significant for the larger dataset (300M), making the system 4X-6X faster than GraphX. The relative inferior performance of the Spark-SQL-Naive shows the inefficiency of executing a recursive query as a series of iterative SQL statements.

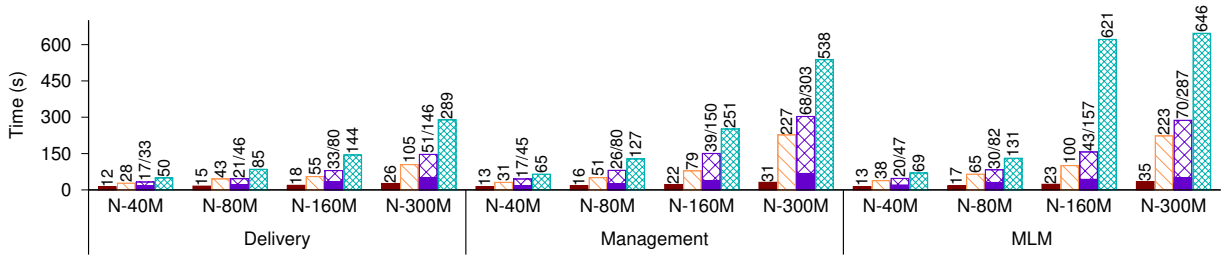


Figure 3.13: Performance comparison on Delivery, Management and MLM queries.

The semi-naive evaluation simulated by Spark-SQL-SN does show some improvements compared to the naive one, by approximately 2X. However, it still lags behind RaSQL by at least 4X, and also slower than GraphX. The result proves that a simple rewriting of the SQL into a semi-naive version will miss many important optimization opportunities in distributed execution such as scheduling, shuffling and caching, thus cannot fully leverage the computation benefits of semi-naive evaluation. Surprisingly, even if we only compare the delta computation (the purple bar) which processes far less data, the Spark-SQL-SN is still slower than RaSQL, which shows the effectiveness of our optimizations.

3.8.3 Scaling Experiments

We report some additional experiment results about how our RaSQL implementation on Spark scales over different cluster sizes, i.e. query execution with different number of workers. We measure the performance of `TC` and `SG` queries on different sizes of synthetic datasets of Table 3.1. As shown in Figure 3.14, our system scales well as 15-worker gains 7X/10X speedups w.r.t. the 2-worker settings on `TC`/`SG`, respectively. The results of `TC` – `G40K`, `SG` – `G10K` and `SG` – `Tree11` using 1-worker are not shown as they either run out of memory or require more than 2 hours to finish.

3.8.4 Performance Comparison with Single-Node Implementation

As a supplement to Figure 3.12, we show the results of GAP Benchmark Suite [BAP15] and COST [MIM15] libraries running the CC query on real world datasets as suggested by

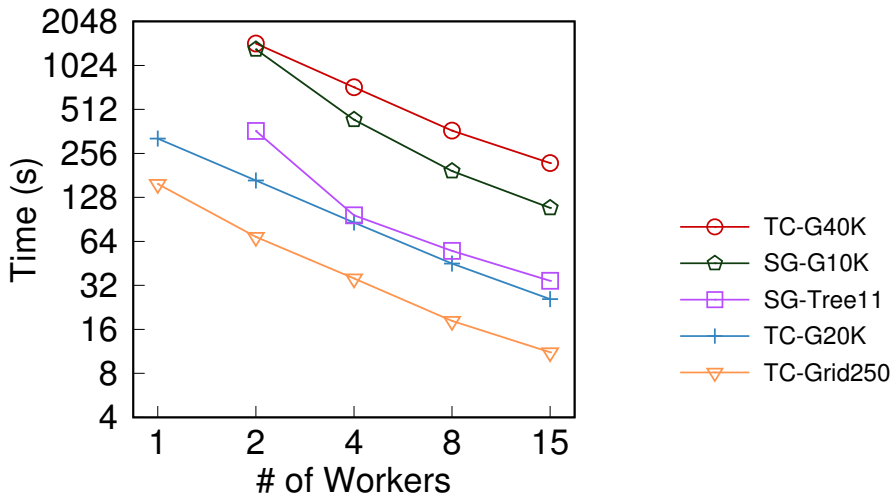


Figure 3.14: Scaling-out Cluster Size

the reviewers. GAP-Serial and COST adopt single-threaded label propagation algorithm, whereas GAP-Parallel executes on 8 cores. As shown in Table 3.3, the two single-threaded libraries tend to execute faster than the distributed systems on small datasets, such as `livejournal` and `orkut`. This is due to their low overhead, but there are other factors as well. In particular, the GAP library is written in C++ and COST in Rust; also, the fact that the COST library reads input in binary format potentially accelerates its execution. However, for larger datasets such as `twitter`, distributed implementations such as RaSQL and Giraph scale up better and clearly win in performance.

Name	`livejournal`	`orkut`	`arabic`	`twitter`
COST	2	<1	14	200
GAP-Serial	13	21	95	763
GAP-Parallel	11	18	93	309
RaSQL	17	19	64	108
GraphX	30	32	72	317
Giraph	21	24	73	106

Table 3.3: CC Benchmark (in seconds)

3.9 Related Work

Providing formal semantics for aggregates in recursive Datalog programs has been the focus of much past research [ZCF97, MSZ14]. In particular, [MPR90] discussed programs stratified w.r.t. aggregate operators whereas [KS91] defined extensions of the well-founded semantics to programs with aggregates, which might have multiple and counter-intuitive stable models. The general approach to deal with all four aggregates proposed in [RS92] uses semantics based on specialized lattices, with each aggregate defining a monotonic mapping in its special lattice — an approach with many practical limitations [Van93]. The optimization of programs with extrema by early pruning of non-relevant facts studied in [GGZ91] and [SR91] exploits various optimization conditions, while the notion of cost-monotonic extrema aggregates was introduced by [GGZ95], using perfect models and well-founded semantics. More recently, [FGG02] devised an algorithm for pushing max and min constraints into recursion while preserving query equivalence under specialized monotonicity assumptions. As shown in [CDI18, ZYI18, ZYD17], the notion of PreM subsumes and simplifies those approaches.

Graph query languages. REX [MIG12] supports a new `WHILE` operator that can be used to express fixpoint computations for SQL statements containing aggregates and other non-monotonic constructs with no guarantee that a formal least-fixpoint semantics is achieved. ScalOps [WCR11] supports a loop construct to allow the relational algebra transformations mixed with the imperative steps. The RACO computation model in Myria [WBB17] provides the relational algebra with iteration extension. Cypher [FGG18], GraphQL [HS08], PGQL [RHK16] and SPARQL [PAG09] are all query languages for property graphs that can express some recursive queries through regular path queries. Graph queries are supported in SQL Server 2017 with the search condition specified by the `MATCH` construct in `WHERE` clause [sql], but the expressive power of this language is quite limited. To the best of our knowledge, none of the languages mentioned above achieves the expressive power of RaSQL and they also fail to provide a rigorous formal semantics for aggregates in recursion.

Large-scale iterative data analytics systems. BigDatalog [SYI16] uses Datalog as its

query language to support data analytics on distributed datasets. Our RaSQL system borrows some of BigDatalog's best practices, such as SetRDD, but uses a new architecture and introduces novel optimizations as shown in in Section 3.6.1 and 3.7. These have delivered huge improvements over BigDatalog as shown in Section 3.8. The SummingBird [BRO14] system provides a domain-specific language in Scala to support online and batch MapReduce computations. The semantics of its aggregation is defined through the theory of commutative semigroups. RaSQL differs from it by providing a declarative language extension over SQL with a rigorous semantics of aggregates-in-recursion guaranteed by the PreM property. The Naiad [MMI13a, MMI13b] system provides a computational model, called timely dataflow, which is effective in parallel processing continuously changing input data. DryadLINQ [YIF08] is a system for data analysis and supports iteration. The Spark stack provides high-level APIs for relational queries [AXL15] and graph analytics [GXD14]. Hyracks [BCG11] is a distributed dataflow engine and supports iterative computation. Graph systems providing a vertex-centric API for graph analytics workloads include Pregel [MAB10], Giraph [CEK15], Powergraph [GLG12], GraphLab [LBG12] and Pregelix[BBJ14]. Distributed Semi-Naive evaluation on MapReduce systems such as Hadoop are discussed in [ABC11, SKH12]. [YZ14] proposed optimized algorithm for parallel recursive query execution on multi-core machines. PrIter [ZGG11] adopts the prioritized execution technique for fast iterative computation.

CHAPTER 4

A Weighted Search Pattern Language and its Efficient Implementation

In this chapter, we will explore another important research field which is the language extensions for pattern matching over large databases and data streams. Recently, there is much interest growing in this area and several large-scale complex event processing systems based on SQL extensions have been proposed and implemented. While these languages and systems are capable of searching complex patterns efficiently in event streams, they lack functionalities that are urgently needed, including results ranking, fuzzy patterns handling, and a specification to clearly express users' preferences in results ordering. To overcome these limitations, we have designed a new Weighted Search Pattern (`WSP`) language for large databases and event streams.

`WSP` extends existing relational sequence languages; thus, it can be used to query relational and semi-structured data in a user-friendly syntax. It also enables applications from various domains such as financial services, genetics, trajectory databases and log analysis. We propose an evaluation model for this language based on *weighted automata*. At the same time, our powerful optimizations on weighted pattern matching make `WSP` quite efficient, enabling its capability to deal with large data in a small amount of time.

	e1	**e2**	**e3**	e4	**e5**	**e6**	e7	**e8**	**e9**	**e10**	**e11**	**...**
Price	**100**	**120**	**130**	120	**110**	**120**	110	**100**	**120**	**130**	**140**	...

Figure 4.1: Price stream (rising period in bold text)

4.1 Introduction

The need for supporting efficiently search of complex patterns in large scale structured and semi-structured data is now well recognized given its wide applicability in areas such as financial market, publish-subscribe systems, RFID-based inventory management, click stream analysis and electronic health systems [ADG08]. The interest in language extensions for pattern matching over data streams is also growing and several research proposals and new SQL standards have been introduced to meet this need. Among them, the SQL-TS [SZZ01] language made the fundamental contributions by introducing simple constructs for specifying complex sequential patterns into SQL, which led to a recently proposed SQL standards SQL-MR [ZWC07]. Based on a similar language syntax, Y. Diao [ADG08] and her group developed a system called SASE using a powerful evaluation model NFA^b, known to be a successful work of performing pattern search over event streams by automata approaches. Recently, Mozafari and K. Zeng proposed a unifying query language and engine called K*SQL [MZZ10b] on structured databases and data streams based on nested words [MZZ10a], which could conduct efficient pattern matching jobs on nested structures such as XML.

These languages and systems extend SQL with abilities that (i) allow a concise definition of complex search patterns, and (ii) are conducive to efficient implementation and query optimization [MZZ12], however, there remain three new challenges in pattern matching over large-scale event streams that have never been addressed before:

Ranking Results. The fact that differences exist between matched results have been ignored by previous works over years. For example, a common task in complex event processing (CEP) is to extract out continuous price rising periods on a price stream (Figure 4.1). Current languages and systems (such as SQL-TS) emit results in the same order as they are matched (left of Figure 4.2). However, for example, some users may be more interested in

R1	[e1 e2 e3]	R1	[e8 e9 e10 e11]
R2	[e5 e6]	R2	[e1 e2 e3]
R3	[e8 e9 e10 e11]	R3	[e5 e6]
...	...	...	...

Figure 4.2: CEP system's ordering vs. User expected ordering

matchings with longer lengths, which means a length-based ranking mechanism will be quite helpful (right of Figure 4.2). Unfortunately, none of the current systems provides such functionalities. In order to recognize most valuable, pertinent and interesting matched results, users have to rank all matchings either manually or resort to other external scripts w.r.t. some criteria, which is inconvenient, inefficient and error prone. The situation becomes even worse in the big data era since current stream processing systems are able to emit gigabytes of results in a relatively short amount of time [MBM09]; thus, a ranking mechanism w.r.t. a proper specification that clearly captures differences between matchings is urgently needed.

Handling Fuzzy Patterns. In scenarios that they are not sure about the details of a pattern before actual results are presented, users tend to use 'fuzzy', 'imprecise' or 'approximate' query patterns under the semantics of current CEP languages. For example, in genomics analysis [SAM03], the search query of finding segment 'ACTG' in a long DNA sequence with inclusions of all possible gene mutations at the second base is written as 'A?TG' ('?' is a wild-card which matches any base). Besides, users may know the fact that the base 'C' is more probable to mutate to 'T' than 'G'. However, such prior knowledge cannot be easily expressed in current pattern search languages, which incurs much extra work if it is utilized to capture distinct traits of fuzzy patterns.

Richer Languages. The languages that need to incorporate users' prior knowledge in pattern description with a support in results ranking are significantly richer than current pattern search languages. For example, they should be able to capture the metrics during the pattern matching process such as how many times one symbol in the pattern are matched against, which value the symbol is matched to and how these metrics are related to users' in-

terest. However, current languages can only describe pattern matching in a boolean way, i.e., matching or mismatching, without considering any kinds of these metrics. Richer languages based on more expressible models are necessary to be invented to satisfy the need.

The goal of our work is to design a new pattern search language and system over event streams that (i) provides a concise and easy-to-understand way to incorporate users' prior knowledge into pattern description, (ii) allows matched results to be automatically ranked according to this rich description, (iii) efficiently emits large number of ranked results in a small amount of time.

Contribution. We achieve these goals through following contributions:
1. We design a new Weighted Search Pattern language (`WSP`) over large databases, which is a natural extension of SQL-TS [SZZ01], and provides versatility with minimal syntactical additions (Section 4.2).
2. We propose a powerful evaluation model for `WSP` based on our study of the features and properties of *weighted automaton* (Section 4.3)
3. We propose various optimization techniques (Section 4.4) for our model to achieve the ability of fast result generation.
4. We develop a prototype system to evaluate our optimization techniques (Section 4.5).

The chapter is organized as follows: the main constructs of our languages are presented in Section 4.2 through some examples, followed by our proposed evaluation model in Section 4.3. The implementation and optimization details are highlighted in Section 4.4. The experiments results are provided in Section 4.5, and a review of the previous related work is mentioned in Section 4.6.

4.2 WSP Query Language

In this section, we briefly introduce our query language `WSP`, which extends the SQL-TS languages. Among all the previous mentioned languages, we choose to make our language

⟨query⟩ ← SELECT ⟨column list⟩
⟨from clause⟩ ⟨pattern clause⟩
⟨where clause⟩ ⟨weight clause⟩
⟨limit clause⟩

⟨column list⟩ ← ⟨derived column⟩ [′,′ ⟨column list⟩]

⟨pattern clause⟩ ← AS PATTERN ′(′ ⟨pattern⟩ ′)′

⟨pattern⟩ ← ⟨atomic pattern⟩ [⟨pattern⟩]

⟨atomic pattern⟩ ← ⟨element⟩ [+|∗]

⟨element⟩ ← ⟨identifier⟩

⟨weight clause⟩ ← WEIGHT ⟨case clause list⟩

⟨case clause list⟩ ← ⟨case clause⟩ [′,′ ⟨case clause list⟩]

⟨case clause⟩ ← ⟨case weight⟩ [WHEN ⟨case condition⟩]

⟨case weight⟩ ← ⟨element⟩ ′ :′ ⟨weight spec⟩

⟨weight spec⟩ ← ⟨number⟩ | ⟨derived column⟩

⟨case condition⟩ ← ⟨column reference⟩ ′ =′ ⟨value⟩

⟨column reference⟩ ← ⟨element base⟩ ′.′ ⟨column name⟩

⟨value⟩ ← ⟨column value⟩

⟨element base⟩ ← ⟨element⟩
| ⟨modifiers⟩ ′(′ ⟨element base⟩ ′)′
| ⟨aggregates⟩ ′(′ ⟨element base⟩ ′)′

⟨modifiers⟩ ← PREV | NEXT | FIRST | LAST

⟨limit clause⟩ ← LIMIT ⟨number⟩

Figure 4.3: Simplified syntax of `WSP`

extensions based on SQL-TS because it provides a succinct syntax and conforms with the standard SQL specification. SQL-TS is also the first pattern search language over event streams that introduces the Kleene-star symbol, an essential structure in WSP, into pattern description.

Extensions made to SQL-TS. We adopt similar constructs to SQL-TS with many language parts inherited from it, including the same syntax and semantics for SELECT, FROM, PATTERN and WHERE clauses. The main extension we made is to introduce a new construct called WEIGHT clause, following the standard WHERE clause, which gives a chance for users to annotate the symbols appear in PATTERN clause with *weight*, a quantitative measurement of a matching's relative 'importance' used in ranking. The detailed semantics of *weights* and other constructs in WSP will be explained in Example 1 with supplements from other examples. A simplified BNF syntax of this language is provided in Figure 4.3, which is sufficient for explanation.

We will start to introduce WSP with an example similar to the genomics analysis case in Section 4.1, where a user wants to search for pattern 'T?G' in a long DNA sequence. He/she has a prior knowledge about the relative high or low probabilities of 'A', 'T' or 'G' that appears in the second base, and incorporates this information into the query using *weights* as shown in Example 1. We also provide a CREATE TABLE[1] clause for each example to show schema of the table used.

Example 1. ***(DNA sequence matching)*** *Find a subsequence in a DNA string with uncertainty in some position.*

Query 1. CREATE TABLE Sequence (pos Integer, base Char(1))

```
SELECT A.pos
FROM Sequence AS PATTERN (A B C)
WHERE A.base =′ T′ and C.base =′ G′
WEIGHT B : 8 when B.base =′ A′,
```

[1] CREATE STREAM if used for event/data streams

```
    B : 4 when B.base =′ T′,
    B : 1 when B.base =′ G′
LIMIT 10
```

Example 1 shows a typical `WSP` query with semantics based on 'immediately follows' relationship between ordered tuples, same as which defined in SQL-TS. The syntax is similar to standard SQL, except for the following structures with new semantics:

- The `AS PATTERN` clause defines a sequential pattern we want to search for. The clause consists of a list of symbols, and each symbol is either a singleton (e.g. `A`) or a Kleene-closure (e.g. $\texttt{A}^*/\texttt{A}^+$), representing matching once or multiple times ($*$ allows arbitrary repetitions, while $+$ requires at least one match) respectively. These symbols can also be referenced in `SELECT`, `WHERE` or `WEIGHT` clauses to support *running aggregates* (e.g. `len`) and *sequence modifiers* (e.g. `first`, `last`, `prev`, `next`), which are detailedly explained in Example 3.

- The `WEIGHT` clause defines how user wants to assign *weights* to matched symbols on different conditions. It consists of a list of `WHEN` sub-clauses that each specifies the *weight* of a symbol on certain condition. In our semantics, *weights* are abstractions of users' preference or importance measurement of a symbol when it is matched against under some condition. For example: `B : 1 when B.base` $='$ `G′` means a *weight* of 1 is generated if a matched tuple represented by B has a value `′G′` in `base` column. The *weights* are arbitrary numbers that can be directly specified by users (Example 1, 2, 3, 4) or derived during the matching (Example 5, 6). The generated *weights* are sum aggregated (refer to section 4.3.3) and a final *weight* is emitted for each matched result, which is used to determine its relative position in the result set. The *weight* of any symbol under any condition is set to *a default value* related to the underlying math structure (refer to section 4.3.3) if not explicitly specified in `WEIGHT` clause.

- The `LIMIT` clause applies the same semantics as it is in standard SQL. Records in the result set are **ranked** based on their *weights* from high to low, so that this clause

helps users focus on a small number of preferred results, which is best suited for some time-critical tasks. This clause is omitted in later examples.

Example 2. *(**Ranking routes**) Find all routes from San Francisco to Los Angeles and rank routes based on number of passed cities and passing through Santa Barbara or not .*

Query 2. `CREATE TABLE Route (id Integer, cname Varchar(20))`

```
SELECT X.id
FROM Route AS PATTERN (X T* Y)
WHERE X.cname = 'San Francisco' and
      Y.cname = 'Los Angeles' and
      T.cname<>Y.cname
WEIGHT T : 10 when T.cname =' Santa Barbara',
       T : −1 when T.cname<>'Santa Barbara'
```

In trajectory analysis, the selection of routes with certain constraints is a popular topic[WZP12]. Here, we show a situation that a user wants to extract all routes from San Francisco to Los Angeles and he/she prefers routes that pass through Santa Barbara and are also short. For this purpose, the *weight* of an intermediate city is set to -1 except for 'Santa Barbara' (set to 10). It is not hard to predict in the result set that routes pass through 'Santa Barbara' and relatively short shall appear at first, followed by longer ones, and then followed by short and long routes that do not pass through 'Santa Barbara'. The relative *weights* (here is 10 and -1) can be adjusted to reflect whether users prefer one versus others ('a specific city' vs. 'route length' in this example).

Example 2 illustrates how users can specify different weights on different conditions for the same Kleene-star symbol (`T` here). `WSP` treats conditions that specified in `WHERE` clause as *hard constraints* and conditions in `WHEN` part of the `WEIGHT` clause as *soft constraints*. Unlike *hard constraints* that every matched tuple must satisfy, tuple that satisfies a *soft constraint* will get specified weight. These weights guide the query evaluation engine to rank results.

For a matching with a smaller weight compared to alternatives, instead of being filtered out completely, they still appear in the result set but are ranked in lower positions. Through this way, users can guide the system to emit results that most pertinent to their needs at first, also leaving room for less interesting results so that both users and systems can save much work on writing and executing a bunch of similar queries.

Example 3. *(**System log analysis**) Find all critical periods, i.e., periods that start and end in severe or fatal errors, ranked by severity.*

Query 3. `CREATE TABLE Log (id Integer, ts Timestamp, tag Char(1))`

```
SELECT first(Q).ts, last(Q).ts
FROM Log AS PATTERN (Q*)
WHERE (first(Q).tag =' s' or 'f') and
      (last(Q).tag =' s' or 'f') and
      len(Q) > 10
WEIGHT Q : 9 when prev(Q).tag =' f' and Q.tag =' f',
       Q : 8 when prev(Q).tag =' s' and Q.tag =' f',
       Q : 8 when prev(Q).tag =' f' and Q.tag =' s',
       Q : 5 when prev(Q).tag =' s' and Q.tag =' s',
       Q : 2 when Q.tag =' f' or 's'
```

We describe a case when a user wants to analyze logs to extract out critical periods, i.e., periods lasting certain time and mixed with 'severe' and 'fatal' level log records[2]. Periods that contain more consecutive 'severe' or 'fatal' level log records mean a higher severity. The pattern (Q*) used here itself cannot accurately reflect users' ideas; thus, it is a vague pattern which should be supplemented by a bunch of hard and soft constraints.

Example 3 shows the power of *modifiers* (*prev*) and *running aggregates* (*len*), which can be used on Kleene-star symbols. The *modifiers* are used to refer to the previous/next tuple of the current tuple that is matched against (*prev*/*next*) or to the first/last tuple of all tuples

[2]'s' means 'severe', 'f' means 'fatal' in example query

that are matched against (*first* and *last*). *Len* is a *running aggregate* that evaluates to the total length of tuple matchings on this symbol; other *running aggregates* such as *sum* and *avg* which sums and averages all values of tuples that are matched against the given symbol.

Example 4. *(**V-pattern Extraction**) Find 'V-pattern's in a stock price fluctuation chart, ranked by steepness.*

Query 4. `CREATE TABLE Stock (ts Timestamp, price Double))`

```
SELECT A.ts
FROM Stock AS PATTERN (A+ B C+)
WHERE A.price < prev(A).price and
      B.price < last(A).price and
      B.price < first(C).price and
      C.price > prev(C).price
WEIGHT A : 1 when prev(A).price − A.price > 10
       C : 1 when C.price − prev(C).price > 10
```

Many existing CEP languages are capable of finding 'V-pattern'[3], however, not all 'V-pattern's found in a stock price fluctuation chart are equally useful. As an illustrative example, users are typically more interested in 'V-pattern's that are steeper (in our example, steepness is defined by number of times the price is more/less than the previous record by a minimal difference of 10), which often indicate a change in the global price variation trend in the near future, while others may just reflect small local fluctuations.

Example 4 shows complex expressions are supported in the *when* sub-clauses, and a fixed weight (`1` here) can be used if users do not care about the contribution of different matched values in ranking, i.e, users want to use 'qualitative' not 'quantitative' measurements in some situations.

Example 5. *(**Salary rising period**) Find periods where an employee's salary is rising,*

[3] a trend of values that go down and up in a consecutive period

ranked by the length of periods.

Query 5. `CREATE TABLE Salary`

`(start Timestamp, end Timestamp, salary Double))`

```
SELECT first(X).start, last(X).end
FROM Salary AS PATTERN (X+)
WHERE X.salary > prev(X).salary
WEIGHT X : (X.end − X.start)
```

Here, one user is interested in the start and end time of periods when his/her salary is rising. Each row in the data represents the user's (unchanged) salary within some time. During the matching process, the weight specified for Kleene-star symbol (`X : (X.end − X.start)` here) will be (sum) aggregated and the weight of a matching will be evaluated to a (maximum) period that the salary is rising.

Example 5 illustrates that *weights* are not limited to pre-defined constant numbers in `WSP`; instead, they can be column references or complicated expressions and their exact values are derived or calculated from previous or current matched tuples. Through this way, users are able to express many queries that are not possible using SQL-TS but `WSP` as shown here and in next example. This example also shows if the *when* sub-clause is omitted in assigning *weight* on a given symbol (`X` here), the specified *weight* value will be counted to any tuple that is matched on this symbol.

Example 6. *(**Active tweet period**) Find active periods over a month when users post tweets, ranked by total length of intervals in that period.*

Query 6. `CREATE TABLE Tweet(id Integer, text Char(500), date Date))`

```
SELECT first(X).date
FROM Tweet AS PATTERN (X+)
WHERE last(X).date − first(X).date > 30 and
    X.date − prev(X).date < 3
```

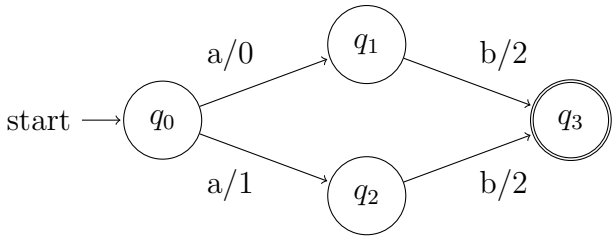


Figure 4.4: Example of a simple *weighted automaton*.

`WEIGHT X` : $-$(`X.date` $-$ `prev(X).date`)

An active period is defined as a period within which consecutive posts' dates do not have intervals of more than three days. Example 6 shows the condition expression in `WHERE` clause can also be listed in `WEIGHT` clause. Here, `X.date` $-$ `prev(X).date` represents the non-posting interval between two consecutive tweets. It is used in `WHERE` clause to satisfy a condition that the interval should not exceed a certain threshold, while it is also used as the *weight* of symbol `X` to ensure the result set is sorted w.r.t. user's preference of periods with small intervals.

4.3 Formal Evaluation Model

After we discuss our language, we introduce a formal evaluation model that we adopt, which is a type of automata called *weighted automata*, powerful and exactly fits the needs for evaluation `WSP` queries.

4.3.1 Weighted Automaton

Our query evaluation model employs *weighted automata* that comprise *finite state automata* and *weights*, and to be more specific, *weights* are *semiring* values attached to transition edges between different states in the automata. Next, we provide the formal definition of a *weighted automaton*:

A **weighted finite automaton** over a closed semiring $S = (K, \oplus, \otimes, \bar{0}, \bar{1})$ is a tuple $A =$

$\langle Q, \Sigma, \delta, q_s, Q_f \rangle$, where

- Σ is a finite alphabet;
- Q is a finite set of states;
- $q_s \in Q$ is the initial state;
- $Q_f \subseteq Q$ is set of final states;
- $\delta \subseteq Q \times \Sigma \times K \times Q$ is a finite set of labeled transitions.

In practice [CDH10], we often relate state transition in *weighted automata* to *finite state automaton* by defining an extra mapping $\gamma : Q \times \Sigma \times Q \rightarrow K$ as a *weight* function (a function that takes a state transition as input and treats the corresponding *weight* as output)

A finite word is represented as $w = \sigma_0\sigma_1...$, where $\sigma_i \in \Sigma$. A *run* over a finite word w on a given *weighted finite automaton* is defined as a finite sequence $r = q_0\sigma_0q_1\sigma_1...q_f$ of states and symbols such that (i) $q_0 = q_i$, (ii) $(q_i, \sigma_i, q_{i+1}) \in domain(\gamma)$ for all $0 \leq i < |w|$. We define the weight $v_i = \gamma(q_i, \sigma_i, q_{i+1})$, and then denote $\gamma(r) = v_0v_1...$ as a sequence of weights obtained by applying mapping γ to consecutive transitions.

A *run* is *accepted* if and only if $q_0 = q_s$ and $q_f \in Q_f$. The *weight* of an *accepted run* is defined by aggregating $\gamma(r) = v_0v_1...v_f$ using $\otimes$ operator in semiring, i.e., $v = v_0 \otimes v_1 \otimes v_2... \otimes v_f$. For a finite word $w = \sigma_0\sigma_1...$, there may be more than one *run* in given weighted automaton that *accepts* the word. We use $\mathbf{v}_i$ to denote the *weight* of ith *run* that *accepts* the word, then the *weight* of this word is defined as $\mathbf{v} = \mathbf{v}_0 \oplus \mathbf{v}_1 \oplus ...$, which is another aggregation using $\oplus$ operator of the semiring structure under the context of this automaton. If we adopt *max-plus semiring* as the underlying math structure, then the *weight* becomes a rational number. The *weight* of a *run* is calculated by using summation over rational numbers, and the *weight* of a *word* equals to the maximum *weight* of any *run* that *accepts* the word. A simple example of this kind of *weighted automaton* is shown in Figure 4.4.

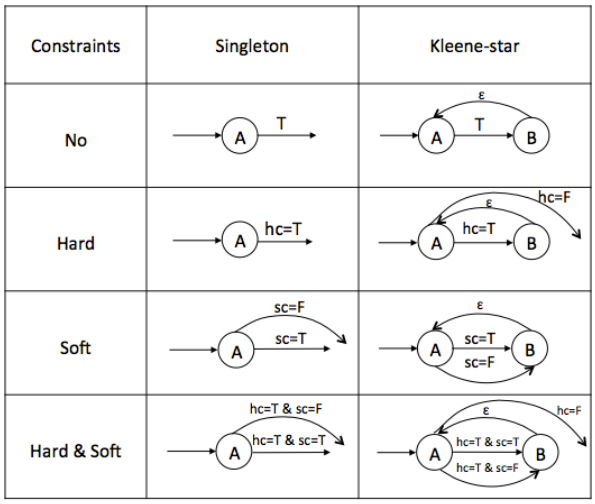


Figure 4.5: Edge Generation Cases

4.3.2 Compilation using Weighted Automaton

We next present our compilation steps for automatically translating `WSP` query into a corresponding *weighted automaton*. The generated automaton will be used for query evaluation over event streams, and is subject to compile-time optimization techniques proposed in section 4.4.1.

Basic Algorithm. We first present a basic algorithm that will faithfully construct an (unoptimized) *weighted automaton* according to our original query.

Step 1. *State Generation*: The `PATTERN` clause in our query uniquely determines the states of the constructed automaton: a singleton symbol is translated to a (singleton) state (without self-loop edges) which consumes exactly one tuple; a Kleene-star symbol $\texttt{A}^*$ is translated to a (recurrent) state (with self-loop edges) which consumes any number of tuples; a Kleene-plus symbol $\texttt{A}^+$ is translated to two states of a singleton state and a recurrent state.

Step 2. *Constraints Classification*: In order to precisely capture transition properties between states, the algorithm classifies the predicates specified in `WHERE` clause as *hard constraints* and conditions that specified in `WHEN` part of the `WEIGHT` clause as *soft constraints*, which conforms with what we have mentioned in Example 2. The algorithm then identifies all the states whose generating symbols (in Step 1) are referred by these constraints and

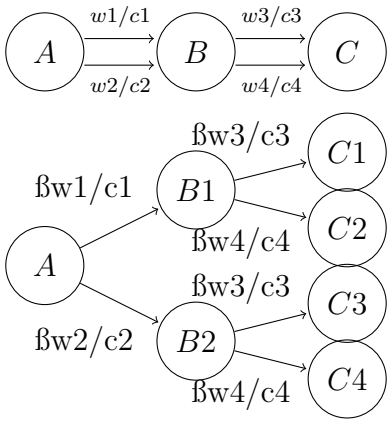


Figure 4.6: Example of Multiple Edges Elimination[4]

assigns correspondingly. After that, all *hard constraints* that related to a particular state are connected into a single conjunctive normal form (CNF). The *soft constraints* remain unchanged since they are disjunctive.

Step 3. *Edge Generation*: For a singleton state, the algorithm adds one edge to its next sibling state for each associated *soft constraint* with the corresponding *weight* attached. Then one more edge representing the case of not satisfying any *soft constraint*s is added with the default *weight* attached. *Multiple edges* are generated if there is at least one *soft constrant*. If *hard constraints* exist, all previous generated edges should also contain them; For a recurrent state, the step becomes a little complicated: the algorithm first inserts an additional singleton state (`B`) between the recurrent state (`A`) and its next sibling state (`C`). The edges from `A` to `B` based on *soft constraint*s are generated similarly to the singleton state case. If *hard constraints* exist, all previous generated edges should also contain them; furthermore, one edge from `A` to `C` (representing rejection on *hard constraints*) and an ε edge (w/o consuming any tuple, to form the self-loop) from `B` to `A` are added. These various cases are clearly classified in Figure 4.5.

Step 4. *Multiple Edges Elimination*: From the very first state, the algorithm checks if there are *multiple edges* between two states. If so, by a duplication of the state that they end with, *multiple edges* can be assigned to different end states. The duplication factor equals to the

[4] w_i/c_i: constraint c_i satisfied and weight w_i gained

number of *multiple edges*. This process is performed recursively on the subgraphs that these duplicated states start with until all *multiple edges* are eliminated. An example of this step is shown in Figure 4.6. Notice this step will often result in a state graph which has a size exponential to the number of symbols, so in practice, we often use another way to eliminate these multiple edges (see section 4.4.1)

After above four steps, a naive *weighted automaton* with its corresponding state graph is constructed. The *weighted automaton* for Example 2 is shown in Figure 4.7.

4.3.3 Evaluation Semantics

In this section, we introduce the evaluation semantics that how a *weighted automaton* conducts pattern matching over event streams. The following procedure adopts the *max-plus semiring* structure with the default weight 0, and is subject to run-time optimization shown in section 4.4.2.

Procedure. (i) The automaton starts from the first state and waits for any incoming tuples; (ii) If a tuple is received by a state, for each of its outgoing edges, the automaton tests the associated *hard constraints* and *soft constraints* to decide whether a transition through this edge is possible. If no edge is satisfied, the automaton backtracks; (iii) For each applicable edge, the automaton calculates the corresponding weight and performs the state transition through that edge; (iv) If the final state is reached, the weight alongside the transitions (from the start state) is *sum* aggregated and an *accepted run* is generated with this weight; (v) If a tuple sequence results in at least one *accepted run*, a *match* w.r.t. the pattern is produced and its weight is determined by the maximum weight among all *accepted run*s. The tuple sequence of this *match* is then added into the final result set with its relative position decided by its weight. The sample matched results obtained by automaton of Example 2 is listed in Figure 4.8.

[5]transition paths/weights also provided, ε transitions omitted

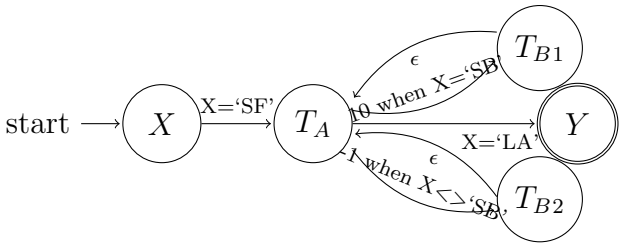


Figure 4.7: Weighted automaton of Query Example 2

Results	**Path**	**Weight**
[SF, SB, LA]	X, T_A, T_{B1}, Y	10
[SF, SB, TO, LA]	X, T_A, T_{B1}, T_{B2}, Y	9
[SF, SC, LA]	X, T_A, T_{B2}, Y	-1
[SF, SJ, SC, LA]	X, T_A, T_{B2}, T_{B2}, Y	-2
...	...	...

Figure 4.8: Sample matched results[5] of Figure 4.7 automaton

4.4 WSP Optimization

Here we briefly introduce our optimization approaches for the *weighted automaton* based on *max-plus semiring* (Section 4.3.1). We classify our various optimization techniques into two main categories: the Compile-time Optimization and the Run-time Optimization.

4.4.1 Compile-time Optimization

During compile time, the compiler will translate the `WSP` query into a *weighted automaton* with states and edges associated with various *soft* and *hard constraints* and *weights* (Section 4.3.2). The optimizations in this phase are mainly related to states reduction and nondeterminism elimination.

States Reduction. If a consecutive number of states that have no *soft* or *hard constraints*

associated, then they can be combined to one state: (i) if states only involve singleton states, the combined state is a singleton state, which consumes number of tuples equals to the number of states combined; (ii) if states involve at least one recurrent state and no singleton states, the combined state is a recurrent state; (iii) if states involve at least one recurrent state and one singleton state, the combined state is a recurrent state, which consumes a minimal number of tuples equals to number of singleton states combined. This optimization is trivial, so we directly apply it on our naive implementation.

Nondeterminism Elimination. Since users can specify *soft constraints* in arbitrary manners as they want, in some cases, two or more constraints that evaluates to different weights will be satisfied simultaneously, which cause nondeterministic transitions. For example, *soft constraints* such as `A : 1 when A.val = 5` and `A : 2 when A.val = prev(A).val` will both satisfy on `A.val = prev(A).val = 5` with weight of `1` and `2` respectively. In the worst case, nondeterminism will result in an exponential number of total transitions. Our optimization is based on *max-plus semiring* semantics: since only the maximum weight among all *accepted runs* is considered as the weight of a matched result, a greedy policy is adopted that conducts only one transition which emits largest weight in nondeterminism cases. Through this way, the exponential number of transitions is avoided and only one answer that equals to the maximum weight of all *accepted runs* (without actually generating them) is calculated out.

4.4.2 Run-time Optimization

Most of the optimizations happen at run-time, which mainly involve early cut-off techniques w.r.t. hard conditions, cache mechanism to reduce number of backtracks, and constraints transformation if `LIMIT` clause is available.

Early Cut-offs. Some *hard constraints* involving *aggregates* can be written in another form to perform early cut-offs. For example, the constraint `len(T) = 8` used on pattern `X T*  Y` is to ensure the matched sequence is of length 10. In the naive implementation, it is used

	e1	**e2**	**e3**	**e4**	**e5**	**e6**	**e7**	**e8**	**e9**	**e10**	**e11**	**...**
Price	**119**	**120**	**130**	**131**	**132**	**140**	130	**120**	**130**	**140**	140	...
Weight	0	1	5	1	1	5	0	0	5	5	0	...
Total	**13**	12	7	6	5	0	0	**10**	5	0	0	...

Figure 4.9: Cache Optimization to reduce backtracks

as a *post-constraint*, i.e., it is only checked after the results satisfying other constraints are generated. Our optimization rewrites it into two parts: `len(T)` $<$ `8` and `len(T)` $=$ `8`, while the former one is checked every time a tuple is matched against the state and the later one is only checked when the former one does not apply. Through this way, any intermediate matchings on `T`* whose lengths exceed 8 are not considered, which eliminates redundant matchings on impossible branches.

Backtracks Reduction. Backtracks result in re-matchings of previously matched tuples. Although it is not totally unavoidable, we can significantly reduce the number of backtracks in some cases. For a recurrent state, every time a matching process starts on this state, it needs to compute weights for any tuple it matches against. However, a lot of re-computations will happen if backtracking happens. For example, if we want to match increasing price sequences in 2nd row of Figure 4.9, all tuple sequences start from `e1` to `e5` and end on `e6` will satisfy the need. However, if we want to employ following *soft constraints*: `A : 1 when A` $>$ `Prev(A)`, `A : 5 when A` $>$ `Prev(A)` $+$ `5`, these matched sequences will produce different weights. The naive way is to backtrack 4 times in order to calculate these weights out.

Our optimization involves a cache of previously computed weights: with a careful look, we find all these sequences involve tuples no more than `e1` to `e6`, and the weight on each tuple remains unchanged regardless of the start and end of the matched sequence; thus, a single calculation of the weight of the first (longest) match from `e1` to `e6` with a cache contains weight of each matched tuple (shown in Figure 4.9, 3rd row) is enough. In this way, we can simultaneously calculate the weights of sequences starting from `e2` to `e5` by subtracting the cached tuple weight consecutively (shown in Figure 4.9, 4th row), without actually conducting the backtracking process.

Type	Weight	Constraint
Strong	10	$T.cname =' Santa\ Barbara'$
Weak	-1	$T.cname <>' Santa\ Barbara'$

Figure 4.10: Constraints Transformation in Example 2

Constraints Transformation. If the `LIMIT` clause is provided, which means users only want to see the top k results in highest weights, we can further conduct optimizations by constraints analysis: *soft constraints* are classified[6] into different categories, e.g., 'strong' and 'weak' based on their relative strength (weight values) and the limit number k. We will see matchings that satisfy 'strong' *constraints* are more likely to be contained in top k results (Figure 4.10). To take advantage of this fact, for any *soft constraints* classified in 'strong' category, we enforce all matched results to obey them, i.e., 'strong' *soft constraints* are temporarily treated as *hard constraints*.

This transformation avoids a large number of matchings due to the filtering effects of the 'strong' *soft constraints*. However, all valid matchings that only satisfy other weaker constraints will be missed. Our solution is to mark all positions in tuple streams (and cache a necessary amount of tuples) from where the probable valid matchings are missed, and perform re-matchings on these positions if not enough results generated. As users tend to specify k in small numbers (typically less than 100) in `LIMIT` clause, only a little extra costs will be incurred if redundant backtracks and re-matchings are kept in minimum by adopting a clever constraints analysis strategy with a combination of previously mentioned cache technique.

4.5 Experiments

The goal of our experiments is to study the amenability of `WSP` queries to efficient execution. We have implemented a prototype system consists of a compiler, an optimizer and a

[6] currently it can only be done on constant numeric weight values

Type	Description
1	Pattern w/o Kleene-star (short)
2	Pattern w/o Kleene-star (long)
3	Pattern w/ Kleene-star (small backtracking len.)
4	Pattern w/ Kleene-star (large backtracking len.)

Figure 4.11: Query Types

query execution engine for `WSP` all in Java. As far as we know, `WSP` is the first language to support ranking on matched results, so we study the overall performance of the prototype system mainly through discussions related to the contribution of our various optimization techniques.

Our experiments were conducted on a 2.27GHz Intel 16-Core Xeon Processor running Ubuntu 12.04, with 16GB of RAM. To demonstrate the performance of our prototype system, we show our results on crude oil price[7] data, which is a real-world dataset contains 1.5 million records.

4.5.1 Optimization Contribution

We used a set of queries that range from type 3 to 4 in Figure 4.11 (small to large average backtracking length) on the crude oil price dataset to show the effects of various run-time optimization techniques in reducing the number of tuple instances visited, including backtracks reduction (cache), early cut-offs (cutoff) and constraints transformation (limit). We first evaluated them in isolation, and then put them together to gain better insight on their separate and combined effects.

In Figure 4.12, we report the number of tuple instances visited over average length of backtracks (on recurrent states before *post-contraint* tested). Let's start with backtracks reduction optimization (red line), which reduces a large portion of tuple visits from the naive

[7]Crude oil Price https://research.stlouisfed.org/fred2

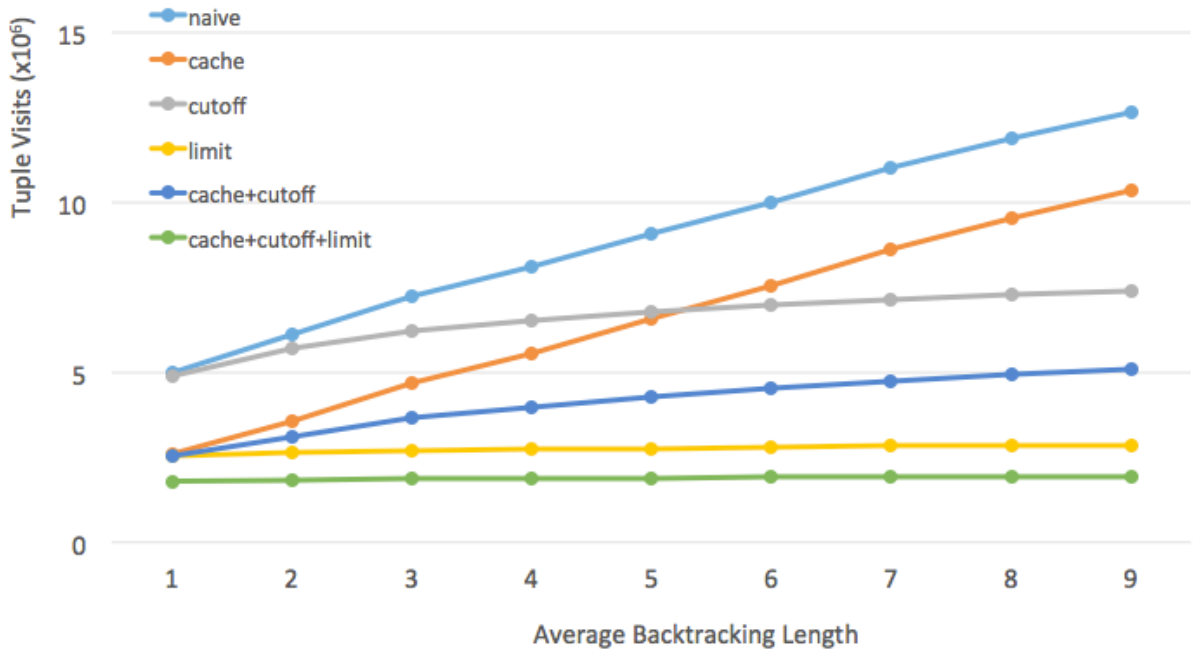


Figure 4.12: Optimization Contribution

implementation. We find the reduced number is not changed with increasing of backtracking length, as it mainly avoids re-matchings of tuples that belong to successful matchings (a constant in our experiment). The optimization of early cut-offs (gray line) will contribute more with the increasing of the average backtracking length since it reduces significant cost in making long backtracks when it becomes a main factor. The combined optimizations reduce more than 60% tuple visits overall.

Constraints transformation optimization (yellow line) is a special optimization since it can only be performed when the `LIMIT` clause is provided by users. The effect is significant if it is applicable[8] (yellow line): it reduces overall tuple instance visits to a small number regardless of the average backtracking length, which is due to a relative small number of matchings and only 'strong' constraints that need to be considered in most time. The combined optimizations (green line) reduce the overall extra tuple instance revisits to no more than 20% w.r.t. total number of tuples in pattern matching process.

[8]we set the limit number k to about 10% of all results in experiment

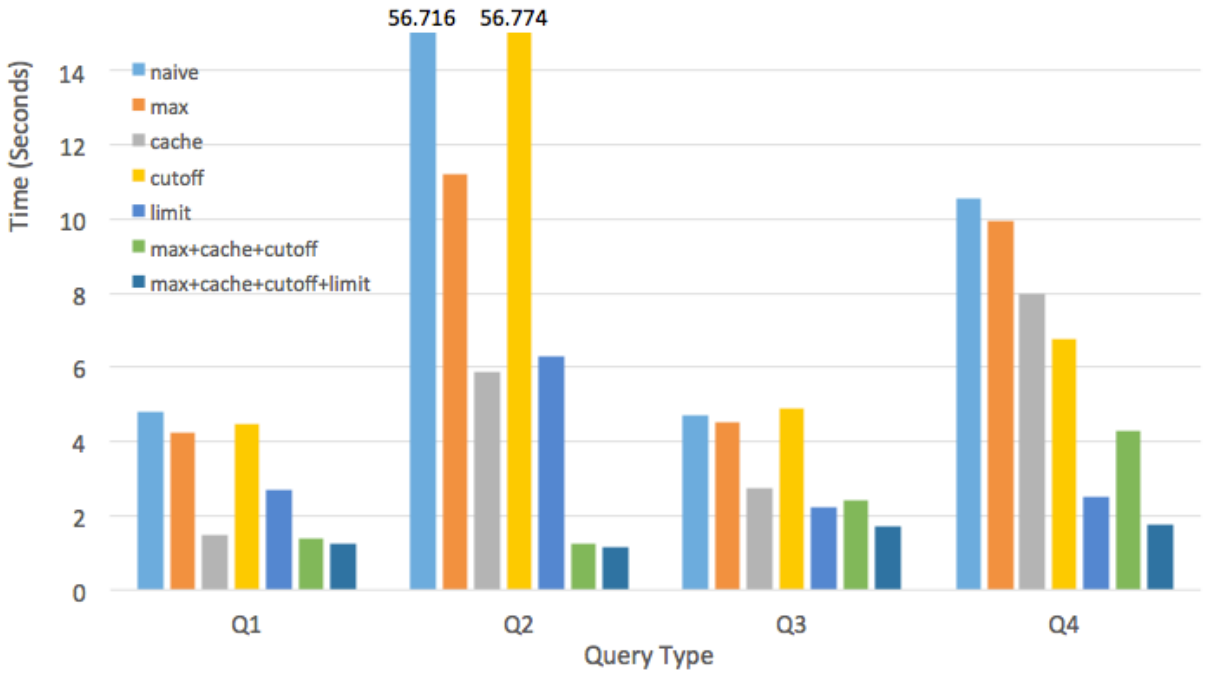


Figure 4.13: Performance Results

4.5.2 Query Execution Time

We measured the performance using four types of queries (one from each type) shown in Figure 4.11 on the crude oil price dataset. To show effectiveness of our optimizations in saving overall execution time, we used various optimization techniques and ran the performance measurement 10 times and averaged the results (Figure 4.13) for each query.

We first look at the results obtained using run-time optimization techniques except the limit. Cache technique contributes most in query type 1,2 and 3 since a lot of weight re-computations are avoided; cut-off technique improves much performance in query type 4 since only long matchings involving Kleene-star provide chances for early cut-offs to work while in other times it only introduces redundant constraint checkings. The compile-time optimization technique – nondeterminism elimination (max) – largely reduces execution time in query type 2 since indeterministic transitions are scaled up exponentially in long patterns. The combined optimizations reduce more than 50% (90% in the extreme case) processing time in average. If limit optimization is considered, our system can further achieve a high processing rate of more than 0.75 million records per second for all types of queries, with a

saving of 80% time in most cases compared to the naive implementation.

4.6 Related Work

Many related works of pattern matching over event streams have already been discussed in Section 4.1. Here, we cover a broader range of topics that related to our work.

Top-k Query Processing. Top-k query processing [IBS08] includes a variety of topics such as top-k selection, join and aggregation. And it also involves various techniques such as approximation[AGP00], randomization[CMN98] and indexing[XCH06]. Most existing top-k processing methods based on the standard relational model [DGK06][IAE03] and SQL[LCI05], with some work[AKM05] touches the semi-structure data domain. To the best of our knowledge, there is no previous work on solving the top-k answer selection problem under the pattern matching semantics. The weight specification combined with `LIMIT` clause in `WSP` provide a way to extract the top-k answers during the search of patterns; thus, it is the first method that extends the top-k answers selection techniques to a broader context.

Weighted Regular Languages. The syntax and semantics of `WSP` are influenced by some theoretical work about weighted regular languages[GTW07], which are regular languages annotated with weights. Their underlying models are also related to weighted automata and recently attract the eyeballs of some researchers[FFG06] since they are more expressible than regular languages. Quantitative languages [CDH09] further generalize all such kinds of languages that can be expressed in the context of weighted automata. However, most works so far have been preliminary in nature and focused on the theoretical feasibility and properties of these extensions. The design of database query language with weighted search capability and, even more so, the design of its underlying implementation, i.e., the connected foci of this report, have not been tackled in the past.

CHAPTER 5

Conclusion and Future Work

In this dissertation, we proposed new extensions to the current SQL standards, which greatly extend the language's expressive power and enable the declarative expression of new applications for complex data analytics tasks that are widely used today.

In the first part, we presented the RaSQL language, a powerful recursive query language with superior expressive power, by making simple extensions to the current SQL standard. Users can express a variety of application queries in RaSQL with a very succinct SQL syntax and rigorous (fixpoint) semantics guarantee. We also show that RaSQL can be compiled into Spark SQL plans with efficient fixpoint operator implementation, which adopts the optimized distributed semi-naive evaluation. Experiments of RaSQL demonstrate the superior performance and scalability on a distributed cluster.

In the second part, we proposed a new weighted search pattern language `WSP` based on *weighted automata*, and implemented a prototype system to evaluate `WSP` queries. Our language provides users with a general and clear syntax and semantics to incorporate their knowledge and preferences in pattern description to help results ranked in desired ways. The experimental results validate the efficiency and capability of our system in processing large numbers of event tuples on a single machine.

Future Work Several exciting research opportunities follow from the RaSQL language. The first opportunity involves optimizing the implementation of RaSQL on various platforms, via indexing, multi-core support and improved partitioning schemes for skewed data. Second, RaSQL's way in handling aggregates in recursion can be extended to support continuous

queries on streaming data, which is an interesting topic focused by recent research [SGZ18]. Third, extending other query languages, such as XQuery or SPARQL, with recursive aggregates is a promising research direction. Furthermore, as SQL extensions to support JSON or other data structures [OPV14, sql] have been proposed recently, combining the benefits brought by their rich data models with the great expressive power of RaSQL represents a compelling research objective. Current work focuses on automating the testing and proving of PreM queries and also on publishing a large library of testing and proving examples that will enable users to take full advantages of PreM when writing their applications.

There are also many exciting areas to explore using weighted semantics on pattern matching over large databases, including but not limited to (i) making constraint transformation optimization capable of handling variables or functions, (ii) extending weighted specifications to nested words to make `WSP` capable of processing semi-structured data such as XML, (iii) introducing new language features so that *weights* can be used to define *constraints*.

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.

[ADG08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. "Efficient pattern matching over event streams." In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pp. 147–160, 2008.

[AGP00] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. "Congressional Samples for Approximate Answering of Group-By Queries." In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pp. 487–498, 2000.

[AKM05] Sihem Amer-Yahia, Nick Koudas, Amélie Marian, Divesh Srivastava, and David Toman. "Structure and Content Scoring for XML." In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pp. 361–372, 2005.

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

[ara] "Arabic-2005." http://law.di.unimi.it/webdata/arabic-2005/.

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

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

[BAP15] Scott Beamer, Krste Asanovic, and David A. Patterson. "The GAP Benchmark Suite." *CoRR*, **abs/1508.03619**, 2015.

[BBB17] David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. "Spanner: Becoming a SQL System." In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pp. 331–343, 2017.

[BBJ14] Yingyi Bu, Vinayak R. Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie.

"Pregelix: Big(ger) Graph Analytics on a Dataflow Engine." *PVLDB*, **8**(2):161–172, 2014.

[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.

[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.

[BRO14] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. "Summingbird: A framework for integrating batch and online mapreduce computations." *PVLDB*, **7**(13):1441–1451, 2014.

[CDH09] K. Chatterjee, L. Doyen, and T.A. Henzinger. "Expressiveness and Closure Properties for Quantitative Languages." In *Logic In Computer Science, 2009. LICS '09. 24th Annual IEEE Symposium on*, pp. 199–208, Aug 2009.

[CDH10] Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. "Quantitative Languages." *ACM Trans. Comput. Logic*, **11**(4):23:1–23:38, July 2010.

[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.

[CEK15] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. "One Trillion Edges: Graph Processing at Facebook-Scale." *PVLDB*, **8**(12):1804–1815, 2015.

[CMN98] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. "Random Sampling for Histogram Construction: How much is enough?" In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pp. 436–447, 1998.

[COK87] Danette Chimenti, Anthony B. O'Hare, Ravi Krishnamurthy, Shalom Tsur, Carolyn West, and Carlo Zaniolo. "An Overview of the LDL System." *IEEE Data Eng. Bull.*, **10**(4):52–62, 1987.

[cyp] "Cypher for Apache Spark." https://github.com/opencypher/cypher-for-apache-spark.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Commun. ACM*, **51**(1):107–113, 2008.

[DGK06] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. "Answering Top-k Queries Using Views." In *Proceedings of the 32nd Interna-*

tional Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006, pp. 451–462, 2006.

[FFG06] Sergio Flesca, Filippo Furfaro, and Sergio Greco. "Weighted path queries on semistructured databases." *Inf. Comput.*, **204**(5):679–696, 2006.

[FGG02] Filippo Furfaro, Sergio Greco, Sumit Ganguly, and Carlo Zaniolo. "Pushing Extrema Aggregates to Optimize Logic Queries." *Inf. Syst.*, **27**(5):321–343, July 2002.

[FGG18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. "Cypher: An Evolving Query Language for Property Graphs." In *SIGMOD*, pp. 1433–1445, 2018.

[flo] "Floyd-Warshall Algorithm." https://en.wikipedia.org/wiki/Floyd_Warshall_algorithm.

[FML12] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. "The SAP HANA Database–An Architecture Overview." *IEEE Data Eng. Bull.*, **35**(1):28–33, 2012.

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

[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." *Journal of Computer and System Sciences*, **51**(2):244–259, 1995.

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

[GM91] Goetz Graefe and William McKenna. "The Volcano optimizer generator." Technical report, COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE, 1991.

[Gra94] Goetz Graefe. "Volcano - An Extensible and Parallel Query Evaluation System." *IEEE Trans. Knowl. Data Eng.*, **6**(1):120–135, 1994.

[gtg] "GTgraph." http://www.cse.psu.edu/~kxm85/ software/GTgraph.

[GTW07] Gösta Grahne, Alex Thomo, and William W. Wadge. "Preferentially Annotated Regular Path Queries." In *Database Theory - ICDT 2007, 11th International*

Conference, Barcelona, Spain, January 10-12, 2007, Proceedings, pp. 314–328, 2007.

[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.

[GZ14] Michael Gelfond and Yuanlin Zhang. "Vicious Circle Principle and Logic Programs with Aggregates." *TPLP*, **14**(4-5):587–601, 2014.

[HS08] Huahai He and Ambuj K. Singh. "Graphs-at-a-time: query language and access methods for graph databases." In *ACM SIGMOD*, pp. 405–418, 2008.

[IAE03] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. "Supporting Top-k Join Queries in Relational Databases." In *VLDB*, pp. 754–765, 2003.

[IBS08] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. "A survey of top-k query processing techniques in relational database systems." *ACM Comput. Surv.*, **40**(4), 2008.

[KLP10] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. "What is Twitter, a social network or a news media?" In *WWW, Raleigh, North Carolina, USA, April 26-30, 2010*, pp. 591–600, 2010.

[KNR11] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing." 2011.

[KS91] David B. Kemp and Peter J. Stuckey. "Semantics of Logic Programs with Aggregates." In *ISLP*, pp. 387–401, 1991.

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

[LCI05] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. "RankSQL: Query Algebra and Optimization for Relational Top-k Queries." In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pp. 131–142, 2005.

[LCY14] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. "Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation." *PVLDB*, **8**(3):281–292, 2014.

[liv] "LiveJournal social network." http://snap.stanford.edu/data/com-LiveJournal.html.

[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.

[MBM09] Marcelo R. N. Mendes, Pedro Bizarro, and Paulo Marques. "A Performance Study of Event Processing Systems." In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*, pp. 221–236, 2009.

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

[MIM15] Frank McSherry, Michael Isard, and Derek G Murray. "Scalability! But at what COST?" In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[mlm] "Multi level Marketing Model." https://en.wikipedia.org/wiki/Multilevelmarketing.

[MMI13a] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. "Differential Dataflow." In *CIDR*, 2013.

[MMI13b] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow System." In *SOSP*, pp. 439–455, 2013.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. "The Magic of Duplicates and Aggregates." In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings.*, pp. 264–277, 1990.

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

[MSZ14] Jack Minker, Dietmar Seipel, and Carlo Zaniolo. "Logic and Databases: A History of Deductive Databases." In *Computational Logic*, pp. 571–627. 2014.

[MUG86] Katherine A. Morris, Jeffrey D. Ullman, and Allen Van Gelder. "Design Overview of the NAIL! System." In *ICLP*, pp. 554–568, 1986.

[MZZ10a] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. "From Regular Expressions to Nested Words: Unifying Languages and Query Execution for Relational and XML Sequences." *PVLDB*, **3**(1):150–161, 2010.

[MZZ10b] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. "K*SQL: a unifying engine for sequence patterns and XML." In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pp. 1143–1146, 2010.

[MZZ12] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. "High-performance complex event processing over XML streams." In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pp. 253–264, 2012.

[Neu11] Thomas Neumann. "Efficiently compiling efficient query plans for modern hardware." *Proceedings of the VLDB Endowment*, **4**(9):539–550, 2011.

[OPV14] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. "The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases." *CoRR*, **abs/1405.3631**, 2014.

[ork] "Orkut network." http://snap.stanford.edu/data/com-Orkut.html.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. "Semantics and complexity of SPARQL." *ACM Trans. Database Syst.*, **34**(3):16:1–16:45, 2009.

[PDB07] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. "Well-founded and stable semantics of logic programs with aggregates." *TPLP*, **7**(3):301–353, 2007.

[pre] "Presto." http://prestodb.io.

[RHK16] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. "PGQL: a property graph query language." In *International Workshop on Graph Data Management Experiences and Systems*, p. 7, 2016.

[RS92] Kenneth A Ross and Yehoshua Sagiv. "Monotonic aggregation in deductive databases." In *PODS*, pp. 114–126, 1992.

[SAM03] Daisuke Shinozaki, Tatsuya Akutsu, and Osamu Maruyama. "Finding optimal degenerate patterns in DNA sequences." In *Proceedings of the European Conference on Computational Biology (ECCB 2003), September 27-30, 2003, Paris, France*, pp. 206–214, 2003.

[SGZ18] Ariyam Das Sahil, M. Gandhi, and Carlo Zaniolo. "ASTRO: A Datalog System for Advanced Stream Reasoning." *CIKM*, 2018.

[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.

[SP07] Tran Cao Son and Enrico Pontelli. "A Constructive semantic characterization of aggregates in answer set programming." *TPLP*, **7**(3):355–375, 2007.

[sql] "MATCH (SQL Graph)." https://docs.microsoft.com/en-us/sql/t-sql/queries/match-sql-graph.

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

[SR92] Divesh Srivastava and Raghu Ramakrishnan. "Pushing Constraint Selections." In *Journal of Logic Programming*, pp. 301–315, 1992.

[SW10] Terrance Swift and David S. Warren. "Tabling with Answer Subsumption: Implementation, Applications and Performance." In *JELIA*, pp. 300–312, 2010.

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

[SZZ01] Reza Sadri, Carlo Zaniolo, Amir M. Zarkesh, and Jafar Adibi. "Optimization of Sequence Queries in Database Systems." In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*, 2001.

[tit] "TITAN Distributed Graph Database." http://espeed.github.io/titandb.

[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 *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pp. 996–1005, 2010.

[Van93] Allen Van Gelder. "Foundations of aggregation in deductive databases." In *Deductive and Object-Oriented Databases*, pp. 13–34. Springer, 1993.

[VRK94] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, and James Harland. "The Aditi Deductive Database System." *VLDB J.*, **3**(2):245–288, 1994.

[WBB17] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. "The Myria Big Data Management and Analytics System and Cloud Services." In *CIDR*, 2017.

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

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

[web] "The RaSQL language." http://rasql.org.

[wor] "WordCount." https://wiki.apache.org/hadoop/WordCount.

[WZP12] Ling-Yin Wei, Yu Zheng, and Wen-Chih Peng. "Constructing popular routes from uncertain trajectories." In *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*, pp. 195–203, 2012.

[XCH06] Dong Xin, Chen Chen, and Jiawei Han. "Towards Robust Indexing for Ranked Queries." In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pp. 235–246, 2006.

[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 *USENIX, OSDI*, pp. 1–14, 2008.

[YZ14] Mohan Yang and Carlo Zaniolo. "Main memory evaluation of recursive queries on multicore machines." In *IEEE International Conference on Big Data*, pp. 251–260, 2014.

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

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

[ZCF10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets." In *HotCloud*, 2010.

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

[ZWC07] Fred Zemke, Andrew Witkowski, and Mitch Cherniak. "Pattern matching in sequences of rows (11)." 2007.

[ZWZ06] Xin Zhou, Fusheng Wang, and Carlo Zaniolo. "Efficient temporal coalescing query support in relational database systems." In *International Conference on Database and Expert Systems Applications*, pp. 676–686. Springer, 2006.

[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 *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018.*, 2018.