Benchmarking Datalog System for Recursive Data Analytics

Johanes Ary

Master of Science

Data Science

School of Informatics

University of Edinburgh

2023

Abstract

Datalog has emerged as a powerful language to do recursive analysis. In this dissertation project, we propose a benchmark based on four real-life applications such as declarative network, data analysis in the knowledge graph, data exchange, and financial computation. This project includes generating network/social graphs and relevant queries in those applications. In order to understand how the Datalog engine processes the query and manages the data, we execute the benchmark over five different Datalog engines: Souffle, BigDatalog, Myria, Bloom, and RecStep. We conclude that our current Datalog engine have the expressiveness to solve the required tasks in various applications. Amongst those engines, Souffle is leading in expressiveness and performs well in most cases. Regarding scalability, Recstep shows promising performance in large datasets but it has limited expressiveness.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Johanes Ary*)

Acknowledgements

The author would like to thank Dr Amir Shaikkha as a supervisor for his guidance throughout this dissertation. The author would also like to thank informatics colleagues for their discussion and insight during the study period in Edinburgh. Finally, the author thanks Indonesia Endownment Fund for Education (LPDP) for sponsoring this completion of the MSc study.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Datalog is a query language that was introduced for the first time in the 1970s. Datalog is well known as a subset of Prolog [11]. In some cases, Datalog can work as a relational database. We may be more familiar with relational databases like SQL as a relational database language because it was first standardised by the ANSI and ISO in 1986 [4].

However, several real-life problems can not be solved using a relational database. If it could, it would require a long time to analyse it. The typical situation is in graph data. That is why there is a different database system, such as a graph database to analyse graph data. Furthermore, SQL has limitations when expressing common graph operations, such as transitive closure in a single query. On the other hand, Datalog appeared with the new generation of applications with high-level abstraction of reasoning and the capability to process complex procedures in large amounts of data [6]. Moreover, there has been a study on Datalog that its capability to support applications in graph data [12, 16], declarative routing [10], program analysis [15, 18], data exchange [13], analysis in knowledge graph [2, 19], and security [3].

Recent studies show us the recent development of the Datalog system and the performance comparison of Datalog systems with the other systems in graph analysis and program analysis [5]. However, there is no clear benchmark of Datalog performance in more complex applications such as declarative networking, data exchange, data analysis in knowledge-graph, and financial computation.

Our project aims to explore the Datalog system's ability to solve real-world problem besides graph analysis and program analysis. We focus on implementing declarative networking, data exchange, and data analysis in graph data. In addition, we also provide

new Datalog queries to solve financial computation with recursive programming. At the end of our project, we will understand Datalog's capability and system performance in solving real-life problems.

1.2 Research Objectives

The main goal of this project is to present clear benchmarking on recent Datalog systems on performing tasks in four different applications, networking, data analysis in the knowledge graph, data exchange, and financial computation. Based on this goal, we can break down the objectives of this project into the following research questions.

1. What kinds of real-life problems can be solved using Datalog?
2. How to write a Datalog program to solve those problems?
3. How do we evaluate recent Datalog systems to perform those tasks?
4. What are the advantage and disadvantages of using Datalog in solving this problem compared with another system?

1.3 Research Structure

This dissertation is divided into five chapters as follows,

In Chapter 1, we explain the problem/opportunity that we have. We mention the goal of our project. We write down several specific objectives to help keep us on track and ensure success.

In Chapter 2, we do a literature review about the recent development of the Datalog engine and the previous implementation of the Datalog engine in various applications. This chapter also highlights our involvement with the Datalog system.

In Chapter 3, we elaborate our approach for this project. We explain how we gather data (synthetic and real data), then continue writing the Datalog program based on the specific task. We specify the system configuration and evaluation metrics used in this project.

In Chapter 4, we show the benchmarking result based on the evaluation metrics mentioned in the previous chapter. We put our analysis regarding the result that we have from this experiment.

Finally, Chapter 5 presents the conclusion and our thoughts on this project. The answers to research questions from Chapter 1 is available in this chapter. Furthermore, we mention our limitations during our work and discuss future work to develop this project.

Chapter 2

Background

2.1 Datalog Basics

Datalog is well-known as a declarative query language system. The syntax in Datalog consists of facts and rules [11]. Facts are representations of the relevant information that we have. Rules are built from our facts to generate a more comprehensive understanding of the facts. For example, we have facts about the parent-child relationship. Then we can represent ancestors as collections of rules from our parent-child information. We have information that Joe is the father of John, and Josh is the father of Joe. From the two facts we have, we create a rule to know who is the grandfather of someone by creating this rule. If X is a parent of Y and Y is a parent of Z, then X is a grandfather of Z. From that rule, we can gather new information that Josh is a grandparent of John.

Facts and rule are represented as Horn Clause [14] as shown in here

$$L_0 :- L_1, L_2, \ldots, L_n \tag{2.1}$$

Where L_n are atoms in the form $R(t_1, t_2, \ldots, t_m)$ such that the predicate symbol R and terms t_n. For atoms on the left-hand side (LHS), we call them the head of a rule, while for the atoms on the right-hand side (RHS), we call them the body of a rule. Terms on the left must appear in the terms on the right to satisfy the safety rule. A term can be a string constant, integer constant, or variable.

The clause that we use to represent facts is unconditional hold. It means there is the head rule but no body rule. The argument we put in facts must be in constant terms (string or integer). In the previous example we can put fact as

$$parent(\text{"}Joe\text{"}, \text{"}John\text{"}).$$

$$parent(\text{"}Josh\text{"},\text{"}Joe\text{"}).$$

It means the relation parent has two tuples, ("*Joe*","*John*") and ("*Josh*","*Joe*"). Moreover, facts could be read from another input sources or Existential Databases (EDB).

A rule has one predicate (or more) in the head of a rule and one or more literals in the body of a rule. The literals could be a predicate, constraint, or negated predicate. For example, if we want to create a rule for grandfather from the facts parent that we have, we can write,

$$grandfather(X,Z) :- parent(X,Y), parent(Y,Z).$$

In this example, grandfather and parent are predicate names while X, Y, and Z are variables. There is no constraint or negated predicated in this example. In the Datalog program, the rule will store the value in IDB (Intentional Database).

Recursion. If there is no recursion in the Datalog program, the syntax has the same understanding as relational algebra in the relational database. Recursive Datalog expresses queries that can't be expressed in a relational database (SQL). IDB predicate P depends on predicate Q if there is a rule for predicate P in the head and predicate Q in the body of a rule. Based on this information, we can draw a graph and call them recursive if there is a cycle in the program. In evaluating a recursive Datalog program, it must have an iterative fixed point. If there are no changes in IDB by applying the rules, it would stop and give us the final result. Here is some example of a recursive query.

$$Ancestor(X,Y) :- Parent(X,Y).$$

$$Ancestor(X,Z) :- Ancestor(X,Y), Parent(Y,Z).$$

In the first rule, we understand that *Ancestor* contains tuples from relation *Parent*. The second rule is the recursive situation where to find the ancestor of Z, we can find it by knowing who is the parent of Y. As long as a tuple(s) satisfies the rule, the relation ancestor would generate new tuples until they reach a fixed point.

2.2 Datalog Extension

In this section, we will discuss semantics and several extensions needed to run all programs for our project. Datalog is a subset of the Prolog program. However, unlike Prolog, Datalog has an additional extension for complex analysis. Negation, constraint, arithmetic/string arguments, and aggregation are essential extensions in Datalog to do more complex analysis in this project.

Negation is the contradiction of arguments. It is symbolized with ¬ or !. For example, in rule $r :- \neg p$, r has value true if p is false, and false if p is true. In Datalog, using negation must be careful that still satisfy safe rules. It means the variable that appears in a negative atom must appear in a positive atom. Second, there is no negation appearing in the head of a rule. Figure 2.1 shows us some examples. The result

```
1 R(X) :- !U(X).
2 R(X) :- U(Y).
3 R(X) :- U(Y), X<Y.
```

Figure 2.1: Unsafe Negation in Datalog Program

(relational R) could be infinite in these three scenarios, even though the relation I is finite. The first rule is not safe because variable X does not appear in positive atom in the body. The second and third rules are unsafe because variable X in the head does not appear in any atom in the body.

Constraint is an atom in the body of rules that gives value true or false. Constraint can be inequality and equality such as $<, >, \leq, \geq, \equiv, \neq$, and string check like containment or matching. By enabling this constraint, it can help determine the IDB to store necessary value based on our rule.

Arithmetic argument can be used to determine the value of terms in IDB that we want to store. For example, $Z = X + Y$ means we want to store value Z based on summation between X and Y. Datalog system should support basic arithmetic arguments such as $+, -, \times, \hat{}, /$, and $\%$.

String argument defines the value of the string that we have. For example, concatenation will give us the output of a string as the combination of two strings. We can express concatenation with function or use arithmetic argument $+$ as an indication of concatenation.

Aggregation, such as sum, count, max, min, and avg, are needed to find some statistics in our data. Aggregation could be found in the recursion or not in recursion.

2.3 Related Work

A previous study shows that the Datalog language can solve problems in various applications, such as program analysis, declarative networking, and graph query. This

section will discuss several Datalog engines that we use in this project. Souffle[1] is a Datalog engine that translates rules to a C++ program to utilize the multi-core machine to evaluate the program. Souffle provides all the extensions that mention in the previous section. Numerous applications have been implemented in Souffle, such as static program analysis [15], security analysis for smart contracts, etc. BigDatalog is a Datalog engine implementation on top of Apache Spark [17]. It outperformed GraphX and native spark queries on large test graphs. It supports declarative queries to do recursive and iterative computations. RecStep is a general-purpose Datalog engine that is built on top of existing parallel in memory of relational database, QuickStep. The latest study about Recstep claims that the system design demonstrate the scalability of their approach [5]. Myria is a database system with imperative language supporting SQL query, iterative, user-defined syntax. Myria is specified for extensive data management and analytics system. Bloom is a programming language for cloud and distributed systems. It supports declarative query language (Dedalus). Dedalus simplifies many challenges in specifying and analyzing program, and it works well on declarative networking [1].

2.4 Our Contribution

In this thesis, we focus on real-life applications for Datalog queries. It will give benefit to assess the expressiveness of the Datalog language. The program in declarative networking represents typical network protocol, such as distance vector, policy-based routing, or finding the best paths. We present common scenarios in data analysis, like getting connection recommendations and company control problems. Furthermore, we provide a new approach to express recursive computation in two technical analyses in finance, simple moving average and auto-regression. The proposed queries can be adjusted to build more complex purposes in the future.

Moreover, we also describe the basic graph data schema representing real-world data distributions. This data generation could be used to build larger graphs without large memory requirements.

We have conducted extensive experiments using large-scale problems. We compare the performance of five current state-of-art Datalog systems.

[1] `https://souffle-lang.github.io/docs.html`

Chapter 3

Statement of Work

IThis chapter explains how we create a query based on the specific task in declarative networking, data analysis in knowledge graph, data exchange, and finance related applications. Then we explain how we gather the data to do our experiment. We also mention our experimental setup and how we measure the performance.

3.1 Datalog Program

This section will discuss about Datalog expressiveness in several applications in different scenario, like declarative networking, data analysis in knowledge graph, data exchange, and financial calculation. The program is written in handful lines of code and declarative language.

3.1.1 Declarative Networking Program

This subsection will discuss several well-known routing protocol, such as distance vector, path vector protocol, and link-state.

3.1.1.1 Path Vector Protocol (PVP) Program

PVP program (Fig 3.1) is used to understand all possible paths from the graph network, and then find the shortest path from source to destination by showing the next hop [10]. A predicate `link` is a fact which contains an entirely directed graph in the form of an extensional database. It has source, destination, and cost. The declaration for predicates is stated in lines 1-4. The first rule (line 6) introduces a predicate `path` as IDB, which is a mirroring of the `link` and defines hop same as the destination. The second rule, line 7,

```
1 .decl link(source:string, destination:string, cost:integer)
2 .decl path(source:string, destination:string, hop: string, cost:
      integer)
3 .decl shortest(source: string, destination:string, cost:integer)
4 .decl nextHop(source: string, destination:string, hop:string, cost:
      integer)
5
6 path(src,dest,dest,cost) :- link(src,dest,cost).
7 path(src,dest,z,c) :- link(src,z,c1),path(z,dest,w,c2), c=c1+c2.
8 shortest(src,dest,c) :- path(src,dest,_,_) ,c = <min> cost : {path(
      src,dest,_,cost)}.
9 nextHop(S,D,H,C) :- path(S,D,H,C),shortest(S,D,C).
```

Figure 3.1: PVP Program

is to find all possible paths by joining the path with the link on the destination from the link equal to the source from the path recursively. IIf they find the tuple that satisfies the condition, it will summate two costs and generate a new tuple with a source from the link and a destination from the path, with a new cost. It will stop when they reach the point where there is no additional path after joining the path with the link. The third rule (line 8) will find the minimum cost based on source and destination. Finally, the fourth rule (line 9) gives us the final path with a minimum cost based on source and destination. In this case, there may be more than one path to reach a destination from a specific source as long as the cost is the lowest.

3.1.1.2 Distance Vector (DV) Program

```
1 .decl bestpath(source: string, destination:string, path:string, cost
      :integer)
2
3 path(S,D,S,C) :- link(S,D,C).
4 path(S,D,P,C) :- link(S,Z,C1), path(Z,D,P2,C2) ,C=C1+C2, P=
      concatenate(S,P2).
5 shortest(S,D,C) :- path(S,D,_,_) ,C = min cost:{path(S,D,_,cost)}.
6 bestpath(S,D,P,C) :- shortest(S,D,C), path(S,D,P,C).
```

Figure 3.2: DV Program

The Distance Vector program (Fig 3.2) is similar to the previous program (PVP).

The main difference is that the DV program produces the path from source to destination instead of a hop. We generate a path by concatenating two strings between the source and the path itself, as shown in the second rule (line 4). Similar to the previous program, the third (line 5) and fourth (line 6) rules will give us a detailed path from source to destination with minimum cost.

3.1.1.3 Policy Based Routing (PBR) Program

```
1 #involve(DV)
2 .decl notpermitPath(source:string, destination:string, path:string,
      cost:integer)
3 .decl permitPath(source:string, destination:string, path: string,
      cost:integer)
4 .decl excludeNode(source:string, destination:string)
5
6 notpermitPath (S,D,P,C) :- path(S,D,P,C), excludeNode(A, W),
      miniPath=concatenate(A,W), contains(miniPath,P).
7 permitPath(S,D,P,C) :- path (S,D,P,C), !notpermitPath (S,D,P,C).
```

Figure 3.3: PBR Program

The PBR program (Fig 3.3) extends the previous program. The PBR program identifies paths from policies set by the network administrator. The program involves the first and second rules from the DV program. Then, in the first rule (line 6), we introduce a new IDB notpermitPath as a collection of paths that the system must ignore. It will ignore any path containing minipath from excludeNode. Finally, the second rule (line 7) uses negation to get our final result. It has the same function as a minus in relational algebra as we need tuples in relation path and not in relation notpermitPath are present. The symbol exclamation mark (!) represents the negation.

3.1.1.4 Cache Routing (CR) Program

The CR program (Fig 3.4) is a variation from policy-based routing. In the previous scenario, we have information on the blocked link. However, we now get the preferable link with better cost value because we store data (cache) to get better data retrieval performance. We trade off capacity for speed.

The first rule (line 3) specifies the path from the link we have on the link but not in linkwithcache. The second rule (line 4) generate all possible path based on it. After that,

```
1 .decl linkwithcache(src:string, dest:string, cost:integer)
2
3 pathbpr (S,D,S,C) :- link(S,D,C), !linkwithcache(S,D,_).
4 pathbpr (S,D,cat(S,P2),C) :- link(S,Z,C1), pathbpr(Z,D,P2,C2), !
      linkwithcache(S,D,_), C=C1+C2.
5 pathbpr (S,D,cat("",S),C) :- linkwithcache(S,D,C).
6 pathbpr (S,D,cat(S,P2),C) :- linkwithcache (S,Z,C1), pathbpr(Z,D,P2,
      C2), C = C1+C2.
7 bestpathCost (S,D,C) :- pathbpr(S,D,_,_), C = min cost:{pathbpr(S,D,
      _,cost)}.
8 bestpathcache (S,D,P,C) :- bestpathCost (S,D,C), pathbpr (S,D,P,C).
```

Figure 3.4: CR Program

the third and fourth rules (lines 5 & 6) add the link from the desirable path linkwithcache, which contains the cache and generates all the possible paths on it. Finally, the last two rules (lines 7 & 8) give us the preferable paths from source to destination with the minimum cost.

3.1.1.5 Multi Cast (MC) Program

```
1 #involve(DV)
2 .decl joinGroup(source: string, destination: string, groupid:
      integer)
3 .decl joinMessage(currentNode: string, prevNode: string, path:
      string, destination: string, groupid: integer)
4
5 joinMessage (C,N,P,D,G) :- joinGroup(N,D,G),bestpathmc(N,D,P1,C),C=
      first(P1),P=shift(P1).
6 joinMessage (C,J,P,D,G) :- joinMessage(J,K,P1,D,G),C=first(P1),P=
      shift(P1),P1!="".
```

Figure 3.5: MC Program

This program (Fig 3.5) shows us another complexity that can be done by Datalog language in a declarative network. MC is used to construct desirable paths from one single/multiple sources to multiple destinations. It involves the rule from the DV program. From that point, the first rule (line 5) will give us the paths P from source to destination that we state in predicate joinGroup. The following rule (line 6) will derive

the paths into next node, C, until it reaches its final destination node, D.

3.1.1.6 Link State (LS) Program

```
1 .decl LinkState(currentNode: string, source: string, destination:
      string, cost: integer, nextNode: string)
2
3 LinkState (S,S,D,C,S) :- link(S,D,C).
4 LinkState (M,S,D,C,N) :- link(N,M,C1), LinkState(N,S,D,C,W), M!=W.
```

Figure 3.6: LS Program

Another protocol that commonly uses is the link-state protocol. This program (Fig. 3.6) gives information on every node to construct a network connectivity map. The initial state in the first rule (line 3) is that each node knows its neighbour's cost. The second rule (line 4) sends the information to their neighbour to create a routing table (flooding). In this case, Variable M is a current node, while variable N is a source of information. Node M construct the connectivity for a whole network by pointing out the next available node that can be accessed from that node or the previously accessible path to node M.

3.1.2 Data Analysis in Knowledge Graph Program

Now, we are discussing some exciting scenarios for data analysis in a knowledge graph.

3.1.2.1 People You May Know (PYMK) Program

```
1 .decl connection(personA: string, personB: string)
2 .decl uconn(personA: string, personB: string)
3 .decl mutualconn(personA: string, personB: string, c: integer)
4
5 uconn(x, y) :- connection(x, y).
6 uconn(y, x) :- connection(x, y).
7 mutualconn (y, z, c) :- uconn(x, y), uconn(x, z), y!=z, !uconn(y, z)
      , c = count:{ uconn(_, z)}.
```

Figure 3.7: PYMK Program

We may be familiar with social network graphs, like Facebook, Twitter, and LinkedIn. In this scenario, we would like to give an example of how social media recommend their users to connect with others. The algorithm behind people you may know (PYMK, Fig.3.7) relies on our understanding of how many mutual connections we have with other people. We can assume that the graph in the predicate connection does not have direction. So in the first and second rules (lines 5 and 6), we generate the connection without direction. The third rule (line 7) does a calculation on the number of mutual friends, c, that variable y has with variable z by self-join predicate uconn. We assume that y and z are not directly connected, as stated in the negation !uconn(y,z).

3.1.2.2 Multi Level Marketing Bonus Calculation (BC) Program

Multi-level marketing is widely used in the selling strategy of various companies. The idea of this scheme is that one person can join by getting an invitation from another member. Every member is obligated to sell the product to another consumer or invite another person to join them in this scheme. In this scheme, the bonus calculation is our sales performance and other people's performance under us. If the people under us (means we are at the root of the tree) have a good sales record, we will get several percentages of additional income. This program (Fig 3.8) replicates the simple bonus calculation formula by considering the number of sales from the people within your network [17].

The first, second, and third rules (lines 13, 14, and 15) create network connectivity for all scheme members. Members at the top of the tree will appear multiple times, depending on how many members are under them. While the lowest member of the tree will appear only once because they do not have any children under them. The bonus calculation will be based on two sales performances. Individual performance is represented by the 5th rule (line 18), and collective performance is represented by the 6th rule (line 19). The 7th rule (line 21) calculates the total bonus by summing those two values. At the end, the eighth and ninth rules (lines 22 and 23) will calculate the net profit after subtracting the bonus from gross profit, `NP = GP - B`.

3.1.2.3 Company Control Problem (CCP) Program

Company control is a classic problem in understanding company behaviour [2]. By ownership information, we would like to understand the relationship between one company and another. Company A has strong influence to company B by having more

```
1  .decl recruit(m: string, n: string)
2  .decl sales(m: string, revenue: integer, profit: integer)
3  .decl memberSales (m: string, sales: integer)
4  .decl rule(ls: integer, us: integer, s: integer)
5  .decl networkMLM(m1: string, m2: string)
6  .decl memberTotalSales (m: string, ns: integer)
7  .decl memberBonusSelf (m: string, b: integer)
8  .decl memberBonusFrontLine (m: string, b: integer)
9  .decl bonus (b: integer)
10 .decl grossProfit (p: integer)
11 .decl netProfit (NP: integer)
12
13 networkMLM (M, M) :- recruit (M, _).
14 networkMLM (M, M) :- recruit (_, M).
15 networkMLM (M, M2) :- networkMLM (M, M1), recruit(M1, M2).
16
17 memberTotalSales (M,SS) :- networkMLM(M,_), SS = sum S : {networkMLM
       (M,NM), memberSales(NM,S)}.
18 memberBonusSelf (M,B) :- memberSales (M,ST), memberTotalSales (M,S),
       rule(LS,RS,BP),S>=LS, S<=RS, B=ST*BP.
19 memberBonusFrontLine (M,B) :-recruit (M,_), B = sum S*BP : {recruit(
       M,NM),memberTotalSales(NM,S),rule(LS,RS,BP),S>=LS, S<RS}.
20
21 bonus(B) :- B = sum B1+B2 : {memberBonusSelf(M,B1),
       memberBonusFrontLine(M,B2)}.
22 grossProfit(P) :- P = sum pp : {sales(_,_,pp)}.
23 netProfit(NP) :- grossProfit(P),bonus(B),NP=P-B.
```

Figure 3.8: BC Program

```
1  .decl own(owner: string, company: string, percentage: float)
2  .decl owns_via(owner: string, midcompany: string, company: string, p
       : float)
3  .decl direct_controls (owner: string, company: string)
4  .decl controls (owner: string, company: string)
5  .decl total_owns_via (owner: string, company: string, n: number)
6
7  owns_via (x,x,y,n) :- own(x,y,n).
8  direct_controls (x,z) :- own(x,z,n), n>50.
9  owns_via (x,z,y,n) :- direct_controls(x, z), own(z,y,n).
10 total_owns_via (x,y,sn):- owns_via(x,_,y,_), sn = sum n:{owns_via(x,
       _,y,n)}.
11 controls (x,z) :- total_owns_via(x,z,n), n>50.
12 controls (x,z) :- controls(x,b), total_owns_via(b,z,n), n>50.
```

Figure 3.9: CCP Program

than 50% share on its company. Not only that, if A controls B, it means A controls all companies under B. The interesting scenario is when A controls B and C, and then B and C partially control company D, 30% and 25%, respectively. In this case, company A controls company D because the ownership combination of B and C is more than 50%.

In the first and second rules (lines 7 and 8) of this program (Fig 3.9), we would like to understand the direct controls of the company. It means the controller holds more than 50% of the company. The third and fourth rules (lines 9 and 10) calculate the indirect control in our ownership relation. Finally, the last two rules (lines 11 and 12) will recursively evaluate the ownership by calculating the total or partial control of the company to another company.

3.1.3 Data Integration and Exchange Program

3.1.3.1 Data Exchange (DE) Program

The simple data exchange is shown in Figure 3.10. From this scenario, we would like to replicate our data in predicate gus into two predicates uBio, and BioSQL [6]. The first rule (line 5) describes that predicate BioSQL directly connects to predicate gus, as it replicates the variable nam and can from gus. The second rule (line 6) shows that BioSQL generates a bid based on gid from gus with the function ord(gid) and then

```
1 .decl gus(gid:integer, nam:string, can:string)
2 .decl uBIO(nam:string, can:string)
3 .decl BioSQL(bid: integer, nam:string)
4
5 uBIO(nam, can) :- gus(gid, nam, can).
6 BioSQL (bid, nam) :- gus(gid, nam, can), bid=convert(gid).
7 BioSQL (b, n) :- BioSQL (b, m), uBIO(m, n).
```

Figure 3.10: DE Program

stores it together with nam. Finally, the third rule (line 7) does recursion to join BioSQL and uBio on nam, so we get the final table consisting of n with a specific bid. If two names have the same bid, it belongs to one full name.

3.1.4 Financial Application Program

This sub-section shows that the Datalog query can do simple calculation methods that require recursion on time-series data. Furthermore, it is possible to perform more advanced calculation techniques by modifying these queries.

3.1.4.1 Simple Moving Average Program

A simple moving average is a standard financial tool for determining stock decisions. With a simple moving average, we can determine if the next future value of a stock is more likely to increase or decrease. The formula of SMA is shown in Eq. 3.1.

$$SMA_n = \frac{A_n + A_{n-1} + \ldots + A_{n-t}}{t+1} \tag{3.1}$$

A_n is the current value, while t is the number of total periods we want to consider in our calculation. A 3-day moving average would average the close price for the first three days as the first-day data point (see Eq 3.2.

$$SMA = \frac{A_n + A_{n-1} + A_{n-2}}{3} \tag{3.2}$$

Given the time series stock price data, we can derive the moving average value using this query in Datalog, as shown in Figure 3.11. We have predicate `salesMA` as historical closing price data for a stock. In the first rule (line 5), we initiate the first value with c equal to one. The second rule (line 6) will add this value to the previous value three times. Finally, the third rule (line 7) calculates the average `avgMA` and stores it in the predicate `resultfinal`.

```
1  .decl salesMA(id:number, sales:float)
2  .decl resultMA(id:number, sales:float, movingAverage:float, c:
       number)
3  .decl resultfinal(id:number, sales:float, avgMA:float)
4
5  resultMA(id,sales,sales,c) :- salesMA(id,sales),c=1.
6  resultMA(id,sales,total,c2) :- resultMA(id,sales,movingAverage,c),
       salesMA(id2,sales2),id2=id-c,c<=3,c2=c+1,total=movingAverage+
       sales2.
7  resultfinal(id,sales,avgMA) :- resultMA(id,sales,movingAverage,c),c
       =3,avgMA=movingAverage/3.
```

Figure 3.11: MA Program

3.1.4.2 Auto Regression Program

```
1  .decl price(id:float, y:float)
2  .decl data(id:float, y:float, lag:float)
3  .decl detail(id:float,mean:float,std:float,cov:float,r:float)
4  .decl finalparameter(b:float, a:float)
5  .decl prediction(id:float, y_predicted:float)
6
7  data(id,y,lag) :- price(id,y),price(idlag,lag),id=idlag+1.
8  detail(id,mean,std,cov,r) :- data(id,y,lag), detail(idd,mmean,sstd,
       ccov,rr), idd=id-1, mean=<eq3.6>, std=<eq3.7>, cov=<eq.38>, rr=<
       eq3.5>.
9  finalparameter(b,a) :- detail(id,mean,std,cov,r),b=<eq3.3>,a=<eq3
       .4>.
10 prediction(id,y_predicted):-data(id,y,lag),finalparameter(b,a),
       y_predicted=a+b*lag.
11 prediction(id+1,y_predicted):-prediction(id,y),finalparameter(b,a)
       ,y_predicted=a+b*y.
```

Figure 3.12: Simpilified AR Program

Another time-series analytic is Auto Regression (AR). This program (Fig 3.12) aims to generate value based on the immediate past value. This work uses the previous timestamp (t-1) to represent auto-regression. Hence, the regression formula will be

$$y = a + bx \tag{3.3}$$

where $y = Y(t)$ and $x = Y(t-1)$.

We could find parameter a and b using Pearson correlation coefficient (r). The formula to get parameter b and parameter a are,

$$b = r\frac{S_x}{S_y} \tag{3.4}$$

$$a = \bar{y} - b\bar{x} \tag{3.5}$$

$$r = \frac{cov(x,y)}{\sigma_x \sigma_y} \tag{3.6}$$

In order to get that coefficient, we need to analyse the mean, variance, and correlation given our data. In a mathematical way, using Welfrod's online algorithm[1], we can compute mean and variance recursively using this formula

$$\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n} \tag{3.7}$$

$$S_n = S_{n-1} + (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n) \tag{3.8}$$

We implement recursive calculation to get that value as shown in the second rule (line 8).

$$C_n = C_{n-1} + (x_n - \bar{x}_n)(y_n - \bar{y}_{n-1}) \tag{3.9}$$

$$C_n = C_{n-1} + (x_n - \bar{x}_{n-1})(y - \bar{y}_n) \tag{3.10}$$

In the third rule (line 9), we get the final parameter by using equations 3.4 and 3.5. Finally, the last two rules (lines 10 and 11) do recursion to get the closing price prediction by substituting the previous closing price with x in the equation 3.3.

3.1.5 Datalog Program and Extension Metrics

We are examining the functionalities of our Datalog programs, and in order to comprehend their operations effectively, we have identified the necessary extensions. The outcomes of this analysis are detailed in Tables 3.1 and 3.2. Across all the programs, the ability to perform recursion has proven crucial, except for the **PYMK** program. The negation statement is utilized in **PBR**, **CR**, and **PYMK**. To ensure the successful execution of these programs, the inclusion of arithmetic arguments is essential, except for **LS** and **DE**. Moreover, the string argument serves a primary purpose in constructing a path in declarative networking. Constraints have been effectively employed to define specific conditions within our programs. Lastly, implementing aggregation functions, such as determining minimum/maximum values, conducting counts, and calculating averages, allows us to tackle problems across all applications proficiently.

[1] https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance

Extension	PVP	DV	PBR	LS	MC	CR
Recursion	v	v	v	v	v	v
Negation			v			v
Artihmetic Argument	v	v	v		v	v
String Argument		v	v		v	v
Constraint	v	v	v	v	v	v
Aggregation	v	v	v		v	v

Table 3.1: Program and Datalog Extension Metrics for Network

Extension	PYMK	BC	CCP	DE	SMA	AR
Recursion		v	v	v	v	v
Negation	v					
Artihmetic Argument	v	v	v		v	v
String Argument						
Constraint	v	v	v		v	v
Aggregation	v	v	v			

Table 3.2: Program and Datalog Extension Metrics for Knowledge-Graph, Data Exchange, and Financial Computation

Dataset	Vertices	Edges	PVP/DV	PBR	LS	CR
G4k	4,000	7,945	157,000	110,000	227,000	174,000
G5k	5,000	12,503	412,000	289,000	679,000	453,000
G6k	6,000	18,011	1,070,000	798,000	2,261,000	1,260,000
G7k	7,000	24,524	2,710,000	2,180,000	6,222,000	3,440,000
G8k	8,000	31,986	6,280,000	5,900,000	15,822,000	8,610,000
G9k	9,000	40,704	13,000,000	14,300,000	40,768,000	18,610,000

Table 3.3: Synthetic Data Gn-p Graph

Dataset	Vertices	Edges	MC	BC	CCP
Tree3	106	106	1,420	1,450	921
Tree4	464	464	6,280	6,470	2,480
Tree5	1,999	1,999	35,200	35,200	13,200
Tree6	7,274	7,274	147,000	151,000	37,900
Tree7	22,400	22,400	458,000	484,000	169,000
Tree8	85,670	85,670	2,300,000	1,880,000	595,000
Tree9	328,549	328,549	9,960,000	4,060,000	1,060,000

Table 3.4: Synthetic Data Tree Graph on MC, BC, and CCP

3.2 Data Gathering

Table 3.3 sshows the synthetic graph used for routing protocol in declarative network experiments. We use these graphs to understand how the Datalog program evaluates **PVP**, **DV**, **PBR**, **LS**, and **CR** on graphs that have specific properties. We use the Erdos-Renyi model to generate these graphs. In the $G(n, p)$ model, the graph is constructed by n nodes and their connection (edges) with independent probability p. The default value of p is 0.001.

Other synthetic graphs are shown in Table 3.4. The tree data (tree-N) represents trees of height N, and the degree of a non-leaf vertex is a random number between 2 and 6. For example, Tree3 and Tree8 are trees of height of 3 and 8, respectively. This type of graph is used for **MC**, **BC**, and **CCP** experiments. As we know in real life, this model is more suitable for our tasks than the previous model.

The real-world graph is displayed in table 3.5. We perform our experiments for **PYMK** not only with the synthetic data but also with the real data. We also performed

Dataset	Vertices	Edges	PYMK
P2P-Gnutella 04	10,876	39,994	1,090,000
P2P-Gnutella 05	8,846	31,839	865,000
P2P-Gnutella 06	8,717	31,525	833,000
P2P-Gnutella 08	6,301	20,777	593,000
facebook	4,039	88,234	2,980,000

Table 3.5: Real Data

Dataset	Initial Tuples	DE
List1k	1,000	2,000
List10k	10,000	20,000
List100k	100,000	200,000
List1M	1,000,000	2,000,000
List10M	10,000,000	20,000,000

Table 3.6: Synthetic Data DE

real data in the network protocol, and we will discuss it in section 4.2.

We assume task **DE** does not require any specific properties for the dataset, so we generate random words with various numbers of tuples in Table 3.6. List1k means we have 1000 tuples with two random words for each tuple.

Finally, for financial calculation data **AR**,**SMA**, Table 3.7 shows historical data for selected companies, such as Apple Inc. (AAPL), Alphabet Inc. (GOOG), The Coca Cola Company (KO), and Tesla Inc. (TSLA) from Yahoo!Finance[2]. We gather the data within the maximum period. It means the data shown in Table 3.7 is from the beginning and is adjusted with the stock split event.

[2] `https://uk.finance.yahoo.com/lookup`

Dataset	Number of Tuples	AR	SMA
AAPL	10754	48,200	64,500
GOOG	4776	30,300	28,600
KO	15506	62,500	93,000
TSLA	3301	25,900	19,700

Table 3.7: Historical Closing Price Data

3.3 Experimental Setup and Measurement

Our experiment is conducted on a single-unit laptop, Lenovo Ideapad Slim 3. In this experiment, we build our programs on Ubuntu 22.04 LTS. It has an Intel i5-1035G1 CPU (1.0 GHz, 4 cores/8 threads). The laptop has 8 GB memory and disk space up to 512 GB solid-state drive.

We evaluate our system with two datasets (synthetic and real), as described in Section 3.2. After we were ready with the query and dataset, we had to decide how to measure time and resource consumption in each run. Our first approach is to start an extensive sequence of stand-alone runs. We ensured that when we ran benchmarking, it was in single-user mode. This way, all the measurements obtained indicated the performance of each query, as separate, as a stand-alone program. We gather time, CPU, and RAM consumption while running each query. However, we decided time consumption as the main performance measure for this benchmarking purpose.

Chapter 4

Result and Analysis

We do our experiment on five different systems, Souffle, Myria, Bloom, RecStep, and BigDatalog, in four different applications, Declarative Network, Data Analysis in Knowledge Graph, Data Exchange, and Financial Computation.

4.1 Correctness of Experiments

To demonstrate the usability of our program, we compare the result with existing solutions. The process begins by using a small dataset for initial testing or exploration. In finding the best path for declarative routing, we compare it with the previous experiment from the previous study [10] and the Python library, networkx[1]. For the knowledge graph, we compare our experiment with related datalog systems, Vadalog [2] and BigDatalog [17]. For data exchange, we took several examples discussed in the previous paper [6]. Finally, for financial computation, we compare the result with the Python library, statsmodel AutoReg[2] for autoregression and rolling function from pandas[3]. The implementation code in Python is represented in Appendix A.3. This phase allows us to familiarize ourselves with the data, understand potential patterns or trends, and identify any initial challenges. Once this initial phase is complete, the next step involves conducting the actual experiment using a larger dataset. The performance analysis of all systems for our programs will be discussed in the next section.

[1] https://networkx.org/documentation/stable/index.html
[2] https://www.statsmodels.org/stable/gettingstarted.html
[3] https://pandas.pydata.org/docs/getting_started/index.html

Program	Souffle	Bloom	Myria	BigDatalog	Recstep
PVP	v	v	v	v	x
DV	v	v	x	x	x
PBR	v	v	x	x	x
LS	v	v	v	v	v
MC	v	v	x	x	x
CR	v	v	x	x	x

Table 4.1: Datalog System Expressiveness in Declarative Network Experiment

4.2 Declarative Network Experiments

For Declarative Network Experiments, we use **PVP**, **DV**, **CR**, **MC**, **PBR**, and **LS** program equivalents in Souffle, BigDatalog, Myria, RecStep, and Bloom. The expressiveness of the Datalog system in evaluating these programs is shown in the Table 4.1.

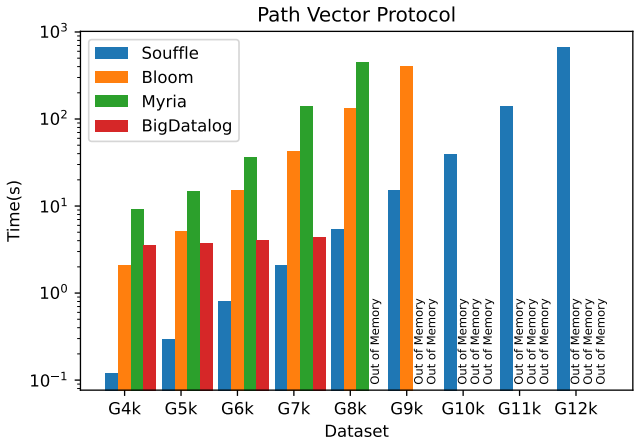

Figure 4.1: PVP

For the **PVP** program, Table 4.1 shows RecStep fails to execute this program. RecStep does not support arithmetic procedures. Summation is needed to calculate the total cost of the selected path. Without summation, we could not find the best path with minimum cost.

Figure 4.1 shows the evaluation time for four systems. After G9k, only Souffle finishes the evaluation. The rest of the systems run out of memory. Souffle is the only system capable of running all graphs in Table 3.3.

Souffle has the fastest execution time on all the datasets. BigDatalog has a steady execution time for all datasets until it runs out of memory on G8k. Bloom has a slightly

better performance than Myria. On the other hand, the execution time for Bloom and Myria increases as the dataset size grows.

We could not perform tests on actual data for this program. They run out of memory every time we try to run the query. The issue relies on the programming logic and data itself. In similar numbers of nodes between synthetic and real data, the real data has more edges than the synthetic does. It means the real graph has different graph properties from the synthetic data. Our initial program can not handle if there is an inner loop in the graph. The real data, as shown in Table 3.5, have more edges and more possibility for an inner loop in the graph. When we run our **PVP** program in small datasets containing a loop, it fails to achieve a fixed point as it has no exit rule. The program runs until there is insufficient memory space and then kills itself.

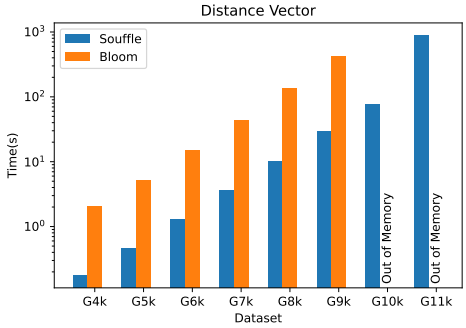

Figure 4.2: DV

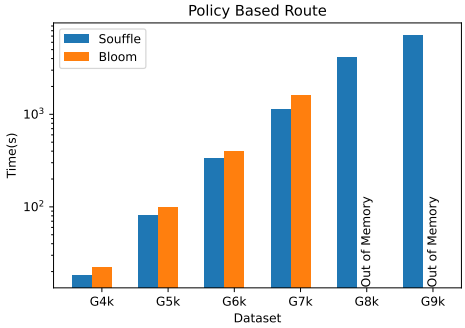

Figure 4.3: PBR

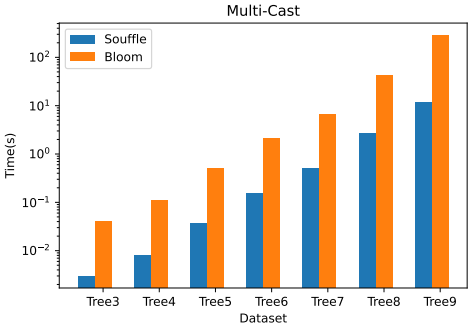

Figure 4.4: MC

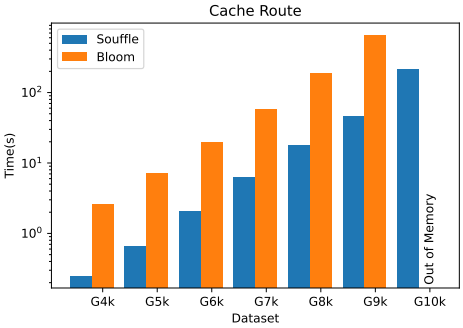

Figure 4.5: CR

For **DV**, **PBR**, **MC**, and **CR**, only Souffle and Bloom can run these programs (Table 4.1). Myria, BigDatalog, and RecStep failed to execute them because they do not support any arguments for the string value. These programs require concatenation to combine two different string values to build a complete path from source to destination.

Figures 4.2, 4.3, 4.4, and 4.5 show the experimental result for these programs on synthetic graphs. From those figures, Souffle outperforms Bloom on all programs and datasets. The MC experiment has been observed to use a dataset that follows a tree structure. It allows us to generate random data for predicate `joinGroup` as we can choose a single source as the root of the tree while we select random nodes as multiple destinations. We can not randomise the source and destination nodes using the Erdos-Renyi graph. We do not have any information to know if they have a connection from one to another.

For real data experiments, we expect to have one or more loops in our graph. We propose a new query that is capable of identifying loops in Figure 4.6. If the existing path contains the source, we have already been there before, so we can ignore it (line 2).

```
1 pathbpr(S,D,P,C) :- link(S,D,C), P=cat("",S).
2 pathbpr(S,D,P,C) :- link(S,Z,C1), pathbpr(Z,D,P2,C2) , !contains(S,
    P2), C=C1+C2.
```

Figure 4.6: DV loop program

However, it does not solve our problem when we run the DV program for entire datasets in Table 3.5. We found that our memory is insufficient to finish the execution. Alternatively, we take the first 1000 rows from Gnutella p2p on 8 August. It generated 8.19 million tuples for 17 seconds in Souffle and 183.817 seconds in Bloom. As stated in Table 3.5, the Gnutella p2p 8 August has 20,777 tuples (edges). The relation between initial edges and generated tuples from the **DV** program is exponential. The other thing that brings on our concern for real data, it generates edges with a long path. Hence, the longer the path, the **DV** program will generate more tuples to find all possible paths from one node to another before finding the best path with minimum cost.

As we can see from Table 4.1, the **LS** runs successfully in all systems. It happens because the link state does not require additional functions or extensions. The **LS** program is uncomplicated, unlike the other tasks that require arithmetic or string arguments like addition or concatenation. The first rule, LS1, states the initial cost from one node to all neighbours. The second rule, LS2, sends information from one node to all destination nodes. Some paper refers to it as a flooding protocol. Hence, it generates more tuples than other programs, as seen in Table 3.3. It also consumes more time to finish the evaluation than other programs.

Figure 4.7 illustrates the performance results in the **LS** experiment. Souffle demon-

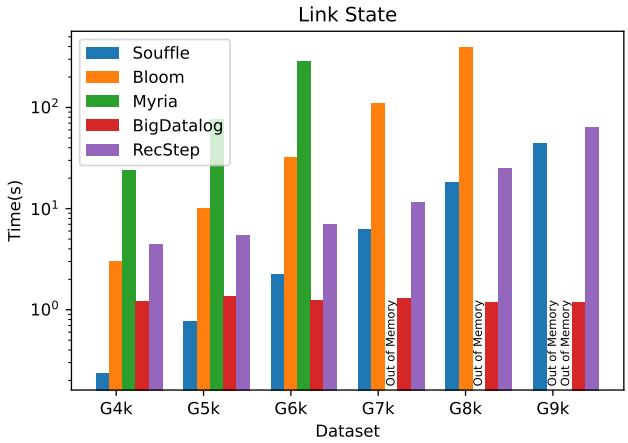

Figure 4.7: LS

Program	Souffle	Bloom	Myria	BigDatalog	Recstep
PYMK	v	v	v	v	x
BC	v	v	v	x	x
CCP	v	v	v	x	x
DE	v	v	v	v	v

Table 4.2: Datalog System Expressiveness in Knowledge-Graph & Data Exchange

strates outstanding performance compared to other systems when dealing with small datasets. However, as dataset size increases, BigDatalog demonstrates better scalability. BigDatalog has consistent performance across both small (1k) and large (10M) datasets, with no significant time difference observed.

4.3 Data Analysis in Knowledge Graph Experiments

Table 4.2 illustrates the Datalog systems' capability to evaluate **`PYMK`**, **`BC`**, **`CCP`**, and **`DE`**.For PYMK program could be run in four systems, Souffle, Myria, BigDatalog, and Bloom but not in the RecStep. RecStep failed to do it because of their lack of ability in the arithmetic argument. It is necessary to understand the total number of our mutual connections.

In terms of performance in Figures 4.9 and 4.8, Souffle is leading. On the other hand, Bloom has poor quality compared to other systems. It is always more than 10x slower for synthetic and real datasets. Similar to the LS experiment, BigDatalog demonstrates its scalability for large datasets. There is no significant time difference between small

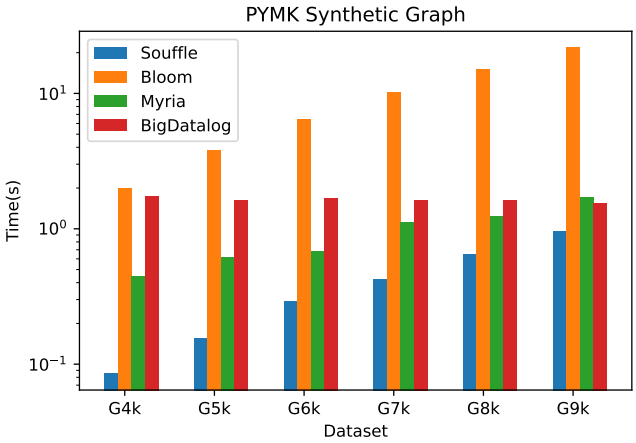

Figure 4.8: PYMK-S

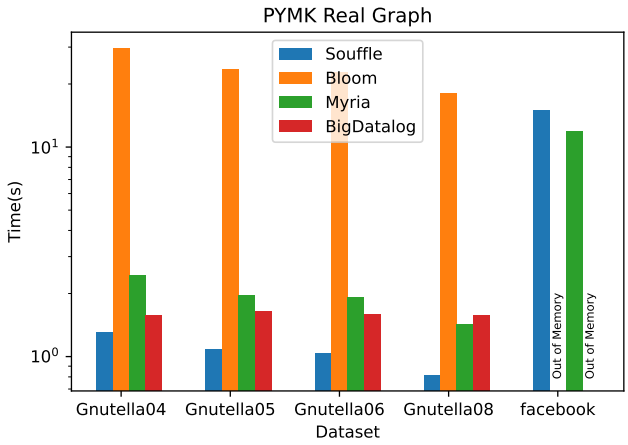

Figure 4.9: PYMK-R

and large datasets, as shown in Figure 4.8. However, when it comes to large dataset, like Facebook, it appears Souffle is better than BigDatalog (Figure 4.9). BigDatalog failed to evaluate the PYMK program in a large dataset.

Figures 4.10 and 4.11 show the performance in evaluating BC and CCP programs, respectively. For the CCP program, Bloom and Myria have similar behaviour, while Souffle is better than those systems. However, in the BC program, Souffle is worse than two other systems from the Tree7 dataset. We suspect that because of the behaviour of this program that requires writing extensive data into an intensional database, Souffle takes more time to do it.

For **`BC`** and **`CCP`** program, only three systems could finish the evaluation. These programs are more complex than previous task, **`PYMK`**, as it needs recursion more than one level. After the recursion, it do some calculation combine with predicate logic

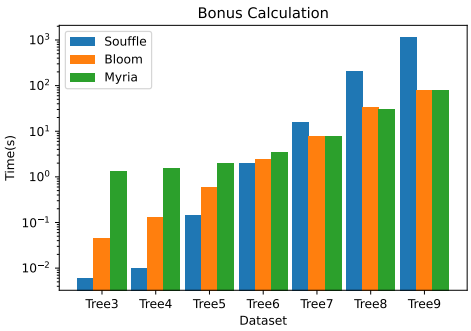

Figure 4.10: BC

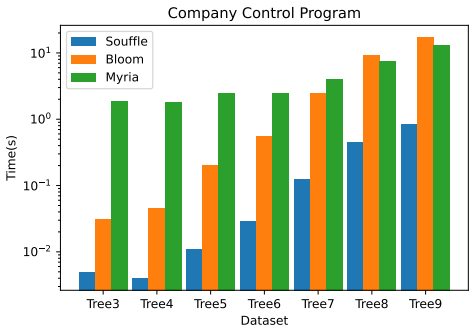

Figure 4.11: CCP

that makes it more complicated. BigDatalog fails to do it. It could not use the existing predicate which is generated from recursion rule to be used in different scenario.

4.4 Data Exchange Experiment

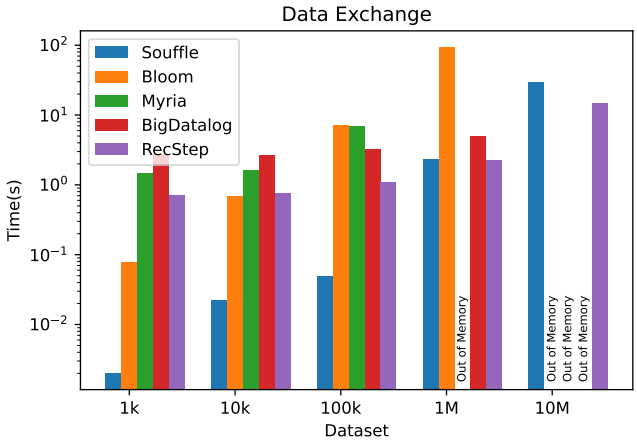

Figure 4.12: DE

Figure 4.12 shows the performance of running DE in five different systems. It shows us that Souffle performs better than four other systems except on the 10M dataset. Recstep performs better than Souffle on the 10M dataset. Similar to the LS experiment, Recstep shows its scalability in large datasets. RecStep's performance is relatively stable in various sizes of datasets.

Bloom performs well in small datasets but not in large datasets. Ultimately, Bloom runs out of memory on the 10M dataset. BigDatalog has consistent time execution before it runs out of memory on the 10M dataset. In comparison, Myria is the only system to fail to execute DE on the 1M dataset.

Program	Souffle	Bloom	Myria	BigDatalog	Recstep
SMA	v	v	v	x	x
AR	v	v	v	x	x

Table 4.3: Datalog System Expressiveness in Financial Computation

4.5 Financial Application Experiments

Table 4.3 shows that Souffle, Bloom, and Myria can express financial computation programs, AR and SMA. BigDatalog is inadequate for scenarios with heavy recursion computing, while Recstep struggles with complex mathematical arguments.

Figure 4.13 and 4.14 illustrate that Souffle performs well in all datasets. Bloom evaluates all programs on all datasets perfectly but with a heavy process. In contrast, Myria only performs well in several datasets for simple moving averages and fails to compute AR in most datasets because it generates massive tuples in an inner predicate.

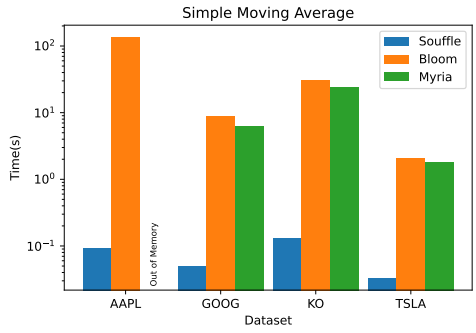

Figure 4.13: SMA

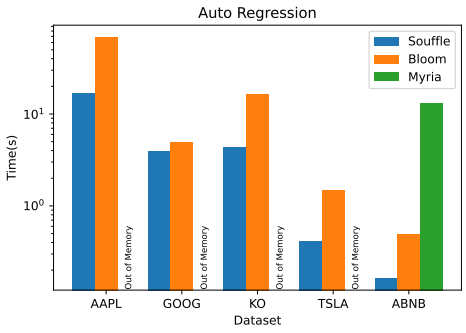

Figure 4.14: AR

Chapter 5

Conclusion

5.1 Summary

Overall, from this experiment, we understand that the Datalog language could solve various problems requiring recursive work, such as declarative networking, data information exchange, data analysis in knowledge-graph, and recursive computing in finance.

We need to understand the basic concept of Datalog, including syntax, rules, extension, and semantics, to write a Datalog program. Then, we need to make sure the program is stratified and safe. There are multiple ways to evaluate our Datalog program. In this experiment, we semantic our Datalog program with a fixed point or naive evaluation.

After experimenting on five Datalog systems, we found that currently, Souffle and Bloom have the capability to perform all tasks. It indicates that those systems satisfy all extensions that we discuss in Section 2.2. While the other systems only have partial functionality that satisfies the extension. Regarding system performance, Souffle could perform better than the other systems. Souffle has C++ translation for their Datalog query to work effectively in a single computer with multiple cores. As the newest Datalog system, Recstep shows its ability to work well in large datasets. In LS and DE experiments, it has a similar or better performance than Souffle in the most extensive dataset. It continues to function properly and without significant differences when dataset size changes. However, it needs future development to handle complex programs requiring mathematical or string arguments.

Overall, writing a Datalog language to solve those problems has benefits in terms of efficiency and performance. Datalog has the expressiveness to perform tasks that

we expect. Hence, we can write complex algorithms with concise queries. In terms of performance, the Datalog systems demonstrate notable efficiency in both time and resource utilization.

5.2 Future Work

We have identified several opportunities for exciting new projects during our work in this dissertation. Regarding the benchmarking project, we suggest doing it in a multi-users scenario. It could give us another perspective on system behaviour in different environments. Seeing the Datalog system works in a cloud or parallel environment will be more interesting. Moreover, the work complexity could be enhanced by finding related applications with complex operations, such as update operators or using dynamic datasets.

Finally, our proposed query can be used for the next benchmarking project or development of the Datalog system. It could be utilized to solve more complex scenarios with some adjustments. For example, in a knowledge graph, we can use it to find a specific pattern that leads to fraud activity in financial transactions [9, 7, 8]. Another example is complex financial recursion computation or another machine learning task, such as logistic regression.

Bibliography

[1] Peter Alvaro, William Marczak, Neil Conway, Joseph M Hellerstein, David Maier, Russell C Sears, William R Marczak, and Russell Sears. Dedalus: Datalog in time and space, 2009.

[2] Luigi Bellomarini, Georg Gottlob, and Emanuel Sallinger. The vadalog system: Datalog-based reasoning for knowledge graphs. 7 2018.

[3] Piero A. Bonatti. Datalog for security, privacy and trust. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 21–36, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[4] Donald Chamberlin. Early history of sql. *Annals of the History of Computing, IEEE*, 34:78–82, 10 2012.

[5] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh Patel. Scaling-up in-memory datalog processing: Observations and techniques. 12 2018.

[6] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5:105–195, 2012.

[7] Dongxu Huang, Dejun Mu, Libin Yang, and Xiaoyan Cai. Codetect: Financial fraud detection with anomaly feature detection. *IEEE Access*, 6:19161–19174, 3 2018.

[8] Xiangfeng Li, Shenghua Liu, Zifeng Li, Xiaotian Han, Chuan Shi, Bryan Hooi, He Huang, and Xueqi Cheng. Flowscope: Spotting money laundering based on graphs.

[9] Xinyang Liu. Fraud detection in non-network knowledge graph, 2022.

[10] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing. pages 289–300. Association for Computing Machinery (ACM), 8 2005.

[11] David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren. *Datalog: concepts, history, and outlook*, pages 3–100. ACM, 9 2018.

[12] Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. Datalography: Scaling datalog graph analytics on graph processing systems. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 56–65, 2016.

[13] Domenico Saccà and Edoardo Serra. Data exchange in datalog is mainly a matter of choice. In Pablo Barceló and Reinhard Pichler, editors, *Datalog in Academia and Industry*, pages 153–164, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] Pascal Schiessle. Datalog-an overview and outlook on a decade-old technology.

[15] Bernhard Scholz, Kostyantyn Vorobyov, Padmanabhan Krishnan, and Till Westmann. A datalog source-to-source translator for static program analysis: An experience report. In *2015 24th Australasian Software Engineering Conference*, pages 28–37, 2015.

[16] Jiwon Seo, Stephen Guo, and Monica S. Lam. Socialite: An efficient graph query language based on datalog. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1824–1837, 2015.

[17] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. volume 26-June-2016, pages 1135–1149. Association for Computing Machinery, 6 2016.

[18] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. pages 515–527. Association for Computing Machinery, Inc, 10 2018.

[19] Victor Vianu. Datalog unchained. pages 57–69. Association for Computing Machinery, 6 2021.

Appendix A

First appendix

A.1 Memory and CPU Utlization

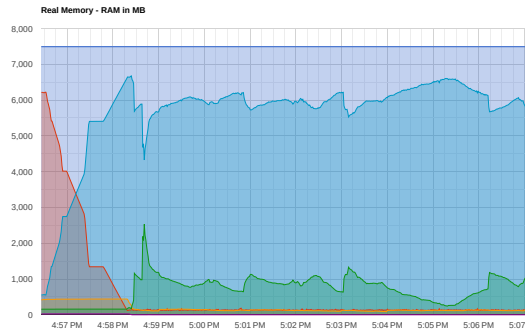

Figure A.1: Memory

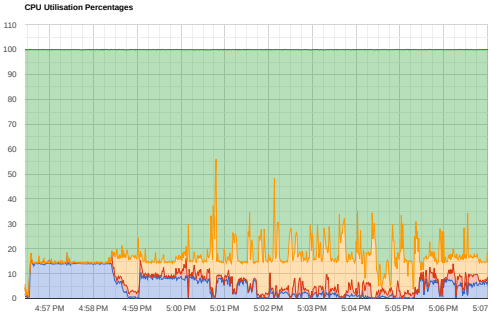

Figure A.2: CPU

Figure A.1 and A.2 indicate the memory and CPU utilization while evaluating PVP program on Souffle. The memory performance goes up to reach 80% of utilization as soon as the program runs, while the CPU fluctuates but still works under maximum capacity.

A.2 Synthetic graph

There are two popular synthetic graphs, ErdosRenyi and Watts. Erdos-Renyi represents the social graph while Watts is more like network graph. However, the more the edges, the more complex the graph we have. In this diverse world, we may see both models in real-life graph.

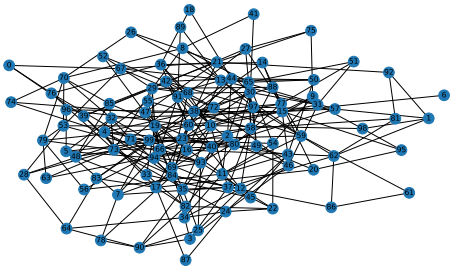

Figure A.3: ErdosRenyi

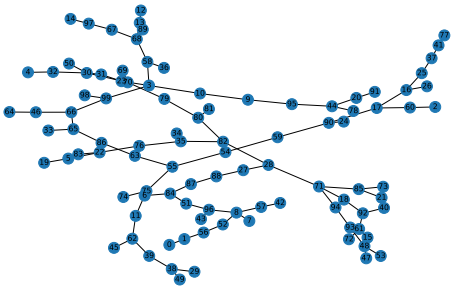

Figure A.4: Watts

A.3 Python Code for Networking and Finance

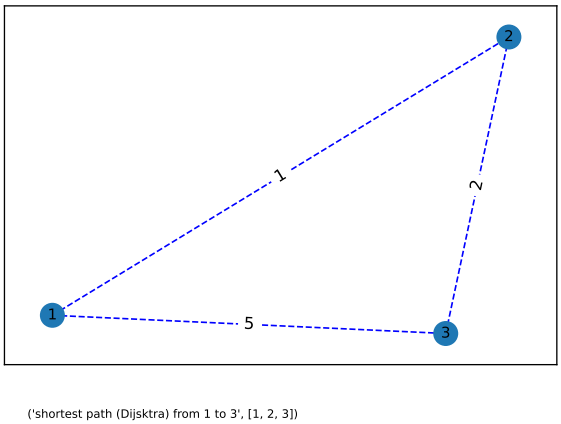

Figure A.5: Network Analysis with Python

```
1  import networkx as nx
2  import matplotlib.pyplot as plt
3  g = nx.Graph()
4  g.add_edge(1,2,weight=1)
5  g.add_edge(1,3,weight=5)
6  g.add_edge(2,3,weight=2)
7  labels = nx.get_edge_attributes(g,'weight')
8
9  pos = nx.spring_layout(g, seed=7)
10 nx.draw_networkx_nodes(g, pos, node_size=200)
11 nx.draw_networkx_edges(
12     g, pos, edge_color="b", style="dashed"
13 )
14 nx.draw_networkx_labels(g, pos, font_size=9, font_family="sans-serif
       ")
15 nx.draw_networkx_edge_labels(g, pos, labels)
16
17 plt.text(-0.5,-1,("shortest path (Dijsktra) from 1 to 3",nx.
       dijkstra_path(g, source=1,target=3,weight='weight')),ha='center',
       rotation='horizontal',va='bottom',fontsize=7)
18 plt.text(-0.5,-1.1,("single source bellman ford",nx.
       single_source_bellman_ford_path(g, source=1)),ha='center',
       rotation='horizontal',va='bottom',fontsize=7)
19
20 plt.savefig('shortest.pdf')
21 plt.show()
22
23 print(nx.dijkstra_path(g, source=1,target=3,weight='weight'))
24 print(nx.single_source_bellman_ford_path(g, source=1))
```

Figure A.6: Network python

```
1  import time
2  start = time.time()
3  import pandas as pd
4
5  # importing numpy as np
6  # for Mathematical calculations
7  import numpy as np
8
9  # importing pyplot from matplotlib as plt
10 # for plotting graphs
11 import matplotlib.pyplot as plt
12 plt.style.use('default')
13 #matplotlib inline
14
15 reliance = pd.read_csv('TSLA.txt', delimiter='\t',)
16 reliance.head()
17 del reliance['Id']
18
19 reliance = reliance['Close'].to_frame()
20 reliance['SMA3'] = reliance['Close'].rolling(3).mean()
21 reliance
22 end = time.time()
23 print ("Time execution, ",end-start,"s")
```

Figure A.7: SMA python

```
1  import time
2  start = time.time()
3  import pandas as pd
4  import statsmodels.api as sm
5  from statsmodels.tsa.api import acf, graphics, pacf
6  from statsmodels.tsa.ar_model import AutoReg, ar_select_order
7  df_AAPL = pd.read_csv('AAPL.txt', delimiter='\t')
8
9  del df_AAPL['Id']
10 df_AAPL
11 AAPL_mod10 = AutoReg(df_AAPL, 1)
12 res_APPL = AAPL_mod10.fit()
13 predict_AAPL = res_APPL.predict(start=1, end=16000)
14 print(predict_AAPL)
15 end = time.time()
16 print ("Time execution, ",end-start,"s")
```

Figure A.8: AR python