THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PERFORMANCE COMPARISON OF THREE DATALOG ENGINES

COREY CAPOOCI
SPRING 2018

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Engineering
with honors in Computer Engineering

Reviewed and approved* by the following:

Gang Tan
Associate Professor of Computer Science and Engineering
Thesis Supervisor

John Sampson
Assistant Professor of Computer Science and Engineering
Honors Adviser

* Signatures are on file in the Schreyer Honors College.

ABSTRACT

Datalog is a lightweight deductive database system that uses a logic programming language in order to construct queries and to update the database [10]. Programmers can implement the deductive database and logic programming language into a single system called a Datalog engine. Researchers are utilizing Datalog engines for problems such as program analysis, information extraction, and network monitoring. Due to a renewed popularity in the Datalog language, it would useful to understand the performance of multiple engines when faced with large datasets. By exposing each engine to the same problems and datasets, the experiment hopes to find a correlation between engine performance and problem size using the execution time of the queries.

The results of the experiment were not entirely consistent. Amongst the three Datalog engines, Soufflé, PA-Datalog, and Datalog Educational System (DES), tested, the results show that Soufflé performs the best for the connectivity algorithm, strongly connected components algorithm, and weakly connected components algorithm. For these algorithms, PA-Datalog and DES performs comparably to Soufflé for some of the less dense graphs. PA-Datalog performs the best for the shortest path algorithm especially for graphs with larger node and density values.

Overall, the results revealed that none of the Datalog engines consistently outperforms the others, but for some algorithms, choosing the correct engine can make an appreciable difference in execution time. Therefore, determining the best Datalog engine is meaningful for certain applications, but hard to predict without initial testing.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank everyone that helped me in the completion of this thesis. I greatly appreciate all the family, friends, advisors, and Schreyer Honors College staff who guided me throughout this process.

First off, I would like to thank my family, especially my parents, for all of their support. Without all of the time and effort that you put into raising and caring for me, I know I would not be here graduating and living the dream. In addition, I would like to extend big thanks to my sister because for everything she has accomplished and will accomplish. I hope I will do the same, hopefully better. Here's to trying to keep up with you.

Next, I would like to thank Professor Tan for all of the mentoring he has provided me. Even though I ask a little too many questions, you were always kind enough to provide me the best answers possible and I could not thank you enough for that. Your mentoring over the past year was priceless, and I cannot thank you enough.

I would also like to thank Professor Sampson for reading over my thesis and for providing the best advice in my academic affairs.

Thanks to those who have been a part of my growing process. All of the friends, coaches, teachers, professors, and neighbors that have touched my life, you provided me the knowledge and motivation to pursue this adventure. I thank you for your support.

Finally, I would like to thank the Schreyer Honors College for giving me the opportunity to try something different and exciting. Without the support from all the faculty and staff, I know this thesis would not have been possible. I extend the sincerest thanks for all that you have done to prepare me for this moment.

Chapter 1

Introduction

Datalog is a lightweight deductive database system with a logic programming language. The programming language provides the means to update and to query the database. Any system that utilizes a deductive database and the logic programming language, Datalog, is a Datalog engine. Engines are created by industry and by research groups in order to solve a variety of different problems from program analysis to network monitoring.

This experiment tests the performance of three Datalog engines, Soufflé, PA-Datalog, and Datalog Educational System (DES). Because Datalog engines are used in so many different applications, it would be useful to know the performance of different engines when faced with large datasets. This experiment evaluates the change in execution times with respect to problem size for each engine. The results give an approximation of which engines perform the best with datasets of increasing size.

To gather data for the experiment, each Datalog engine executes four graph algorithms on graphs with predetermined characteristics, the number of nodes and the graph density. The characteristics of the graphs were chosen so that the graph algorithm could be run within a 30-minute time limit. There are 20 different graphs types, and each graph type is randomly generated three times. Each randomly generate graph is inputted into each Datalog engine for execution on the appropriate graph algorithms. The time to output the results of the graph algorithms is the execution time.

To provide a better understanding of the project and the technology it uses, this thesis includes a background of the Datalog language, the Datalog engines, and the graph algorithms. Afterwards, the experiment design and methodology section discusses the problems associated with this experiment and the solution to the problems. It discusses in detail the layout of the experiment and the generation of random graphs. Finally, there is a discussion of the results, and a conclusion analyzing the significance of the results and the next steps in the research of performance in Datalog engines.

Chapter 2

Background

The experiment tests the performance of Datalog engines in comparison to problem size. To draw a comparison, each Datalog engine executes four graph algorithms on randomly generated graphs. The execution times of these algorithms provide insight into the speed of the Datalog engines. This section reviews the Datalog language, the Datalog engines, and the graph algorithms.

Datalog

Datalog is a query language for deductive databases. The language queries a database by creating rules and executing known facts on those rules. The rules create more facts and continue to add facts to the database until no more facts can be derived. The rules are based on the information in the database. For example, if a graph is stored in the database, the user can store edges and nodes as known facts and write rules that find the paths between individual nodes based on the edges in the graph. Datalog has a variety of applications including data integration, declarative networking, program analysis, information extraction, network monitoring, security, and cloud computing [7].

A Datalog engine is software that implements the concepts of Datalog. Private sector and universities develop Datalog engines. Some versions of Datalog are open-source and others, such as LogicBlox, are for-profit. Nevertheless, all of the Datalog engines allow the ability to model problems and query results with a Datalog language.

The basis of the Datalog system is the predicate, or relation. In Datalog, there are two kinds of predicates, extensional database predicates (EDB) and intensional database predicates (IDB). The EDB are the known facts in the program. The EDB do not change as the Datalog program is executed. The rules defined by the programmer create the IDB. As the program is running, the rules generate IDB, which could generate more IDB and so on. The Datalog program will continue until no more IDB are generated.

To use Datalog, the programmer programs the rules and the facts into files. For the Datalog engines used in this experiment, the syntax for defining a rule is <head> <operator> <body>. The head and the body are a list of predicates. If all of the predicates in the body exist, then the rule generates the predicates in the head as IDB. The operator varies depending on the engine, but for the engines used in this experiment the operator is either ":-" or "<-". The operator delineates the head from the body and dictates that the predicates in the body imply the predicates in the head.

Figure 1 shows a simple Datalog program. In this example, X, Y, and Z are variables while 1, 2, 3, 4, and 5 are constants. The first two lines illustrate how to define a path in a graph. The first rule states that if there is an edge from node X to node Y then there is a path from node X to node Y. The second rule states that a path exists from X to Z if there is a path from X to Y and an edge from Y to Z. The bottom three lines are facts stating that there are edges from 1 to 2, from 3 to 4, and from 4 to 5. The edges are the facts that are initially inputted into the program, aka the EDB, and the

```
path(X,Y) :- edge(X,Y).
path(X,Z) :- path(X,Y), edge(Y,Z).

edge(1,2).
edge(3,4)
edge(4,5).
```

Figure 1. Example Datalog Code

paths are the facts that are generated by the program, aka IDB. In a Datalog program, this is an important distinction because Datalog will not generate EDB.

Datalog Engines

Datalog engines are systems that implement a deductive database system and use Datalog as a query language. This section breaks down the engines used in this experiment. Each section gives a background of the engine, goes over the steps to execute a query, and comments about installing the engine

PA-Datalog

PA-Datalog is a Datalog engine based on a modified LogicBlox v3 engine [12]. The Programming Languages and Advanced Technologies group at the University of Athens maintains PA-Datalog. PA-Datalog requires a Linux system such as Ubuntu and the following libraries: tcmalloc, protobuf, Boost, and CppUnit.

PA-Datalog provides a few features, such as allowing multiple databases, that make it more practical in comparison to the other engines. PA-Datalog allows the user to create multiple databases and add different rules to each of the databases. Therefore, the user can run the same facts on multiple different databases, with different rules, without destroying or overwriting the contents of a previous database. In addition, each database becomes permanent after its creation and can be updated throughout its lifetime.

PA-Datalog provides command line tools in order to modify and query the database. To use PA-Datalog, there needs to be two files, the facts file and the rules file. In the facts file, the

programmer can add or remove an EDB from the database. In the rules file, the rules are defined and all of the predicates are declared with the arguments that they take. These arguments must have defined types such as int, uint, string, etc. 'bloxbatch' commands are used to create and modify the contents of the database. To run a query in PA-Datalog, first, use the 'bloxbatch' command to create the database. Next, add a block of rules to the database. Then, execute the facts on the database with the rules added. Finally, use the print command to query the result. The times to execute all of these steps are summed together for the total execution time.

Soufflé

Soufflé is a Datalog engine developed by Oracle Labs in Brisbane for use in program analysis [16]. Soufflé's website contains prebuilt packages for Debian and for Mac OSX operating systems. In order to install Soufflé, packages and tools such as Make, Autoconf tools, GNU G++ supporting C++ 11 and OpenMP, Bison, Flex, and Doxygen must be on the operating system. After installation, use the 'souffle' command with a Datalog file as an argument to execute a query and to output the results to the command line.

Soufflé provides a multitude of features to improve the execution time of queries. The Datalog engine provides an interpreter, compiler, and feedback-directed compilation infrastructure for compiling and executing Datalog programs. The compiler compiles the Datalog source into a C++ program, which may improve the performance. Soufflé also allows for parallel execution in effort to improve execution time. This experiment only uses the interpreter with no parallel execution.

Soufflé has a similar user experience in comparison to PA-Datalog, but Soufflé only requires running one command instead of four. In addition, Soufflé requires the use of one or more files. In this experiment, all of the facts and rules are kept in the rules file. Although, Soufflé allows the use of a rules file and multiple EDB files.

Datalog Educational System

Datalog Educational System (DES) is a Datalog engine developed for educational use of deductive database systems [4]. DES is implemented in Prolog and includes other languages besides Datalog such as SQL, Relational Algebra, Tuple Relational Calculus, and Domain Relational Calculus. DES's website provides downloads for Windows, 32-bit Linux, and Mac OSX. For other systems, such as 64-bit Linux, there are instructions to create an executable specifically for the operating system. For this experiment, the executable was created with SWI-Prolog.

DES's user experience differs from PA-Datalog and Soufflé because it provides a user interface within the command line. With the user interface, the programmer can consult a file, essentially the add block command in PA-Datalog, and execute a query. Therefore, all of the facts and rules must be stored in a single .dl file. After consulting a .dl file, the user can modify the database through the command line. In contrast to Soufflé and PA-Datalog, the predicates do not need to be defined in the file and the types of the arguments do not need to be explicitly defined in the system.

Graph Algorithms

Graph algorithms are an easy-to-implement and scalable method of analyzing performance amongst different Datalog engines. This section names and explains the graph algorithms in the experiment. It also gives a brief explanation of the algorithm in the Datalog code. Short snippets of code are shown as examples. The full code used in the experiment is available in Appendix E for PA-Datalog, Appendix F for Soufflé, and Appendix G for DES.

Connectivity

```
path(x, z)  <- edge(x, z, _), node(x), node(z).
path(x, z)  <- path(x, y), edge(y, z, _), node(x), node(y), node(z).
```

Figure 2. Connectivity Code

Figure 2 shows the graph connectivity algorithm written in PA-Datalog. Connectivity describes whether there exists a path between two nodes in a graph. The algorithm works by initially defining a path as any edge between two nodes. Therefore, each edge predicates creates a path predicate between the same two nodes. Furthermore, paths are recursively defined such that if there exists a path between node x and node y and there exists an edge from node y to node z, then there is a path from node x to node z. A practical application of the connectivity algorithm includes analyzing reachability in program analysis. In this application, blocks of assembly instructions are considered nodes and jumps or branches between blocks are considered edges.

Strongly Connected Components

```
path(x, z)  <- edge(x, z, _), node(x), node(z).
path(x, z)  <- path(x, y), edge(y, z, _), node(x), node(y), node(z).
scc(x, y) <- path(x, y), path(y, x), x < y.
scc_g_init(x, s) <- scc(x, _), !scc(_, x), s = x.
scc_g(x, s) <- scc_g_init(x, s).
scc_g(x, y) <- scc_g_init(x, s), scc(s, y).
```

Figure 3. Strongly Connected Components Code

For two nodes, a and b, to be strongly connected components, there must exist a path from node a to node b and vice versa. Furthermore, in a strongly connected subgraph, for every node c in a subgraph A, there exists a path from c to every other node in the subgraph A and a path from every other node to node c. One application of the strongly connected subgraph algorithm is to improve the performance in profilers [6].

Figure 3 shows the strongly connected components algorithm written in PA-Datalog. The algorithm instantiates all of the paths in the graph. Then, the code instantiates all of the strongly connected components by finding the nodes x and y such that there exists a path from node x to node y and vice versa. In the scc_g_init rule and the scc_g rule, the left argument holds the name of the strongly connected subgraph. The name of the strongly connected subgraph is the value of the node with the lowest value in that subgraph. Therefore, for a strongly connected subgraph with nodes 2, 4, and 15, each node is in the strongly connected subgraph with the label of 2. These facts generate the predicates scc_g(2,2), scc_g(2,4), and scc_g(2, 15). To create the strongly connected subgraphs, first, the rules find the smallest value node in each strongly connected subgraph. Next, this value becomes both arguments in the scc_g_init predicate. Then, each of the nodes n in the strongly connected subgraph create predicates scc_g(s, n) where s is the smallest value of the node found earlier.

Weakly Connected Components

```
path(x, z)  <- edge(x, z, _), node(x), node(z).
path(x, z)  <- path(x, y), edge(y, z, _), node(x), node(y), node(z).
wcc(nd, nd)   <- node(nd).
wcc(nd1, nd2) <- path(nd1, nd2), nd1 < nd2.
wcc(nd1, nd2) <- path(nd2, nd1), nd1 < nd2.
wcc(nd1, nd3) <- wcc(nd1, nd2), wcc(nd2, nd3).
wcc(nd1, nd3) <- wcc(nd1, nd2), wcc(nd3, nd2).
wcc_g[nd2] = nd  <- agg<<nd = min(nd1)>> wcc(nd1, nd2).
```

Figure 4. Weakly Connected Components Code

Weakly connected components in a directed graph exist if there are two nodes x and y such that there exists any series of edges between x and y. Weakly connected subgraphs are subgraphs where if all of the directed edges are converted to undirected edges, then for every node z in the subgraph B there is exists a path between z and every other node in B. Some applications of weakly connected components include analyzing the nodes in the backbone of an ad-hoc network, and improving text-based search engines in locating high-quality pages [5, 8].

Figure 4 shows the weakly connected subgraph algorithm written in PA-Datalog. The wcc predicate indicates all the nodes that are weakly connected. The "wcc_g[nd2] = nd" rule states that node nd2 is in the weakly connected subgraph labelled nd. Much like the strongly connected subgraph algorithm, each weakly connected subgraph is labelled by the node in the subgraph with the smallest indicator value. For example, if there is a weakly connected subgraph with nodes 3, 5, 6, and 15. All of these nodes would be in the weakly connected subgraph labelled 3. With this subgraph, the predicates wcc_g[3] = 3, wcc_g[5] = 3, wcc_g[6] = 3, and wcc_g[15] = 3 are generated. To find all weakly connected subgraphs, the rules generate all possible weakly connected subgraphs. Then, for each node in the graph, the aggregate

function finds the weakly connected component with the smallest node value. This smallest value represents the weakly connected subgraph that the node resides in.

Shortest Path

```
path_dis(n,n,0,0)            <- node(n).
path_dis(src,dest,dist,rnd1) <- path_dis(src, n, dist1, rnd), edge(n,dest,dist2),
                                num_nodes(t), dist=dist1+dist2, rnd1=rnd+1, rnd<t.

min_path[src, dest] = d      <- agg<<d = min(dist)>> path_dis(src, dest, dist, _).
```

Figure 5. Shortest Path Code

The shortest path algorithm for a weighted directed graph determines the path with the smallest weight between two nodes. The weights of a path are the sum of the edges between the nodes. Some of the applications of the shortest path algorithm include networking, where the routing algorithms use a shortest path algorithm to route packets, and travel, where the shortest path algorithm determines the cheapest way to travel via plane between source and destination airports.

Figure 5 shows the shortest path algorithm written in PA-Datalog. In this algorithm, the path_dis predicate holds distances between a source node and a destination node. In order to ensure termination of the execution of the program, the number of edges between source and destination nodes cannot exceed the number of nodes in the graph. This algorithm prevents the program from entering an infinite loop due to cycles in the graph. After all of the path distances are calculated, the “min_path[src, dest] = d” rule calculates the shortest path from src to dest using an aggregate function. The aggregate function examines all path_dis predicates with src as the source node and dest as the destination node and chooses the smallest dist, or path weight.

Chapter 3

Experiment Design and Methodologies

Many of the challenges in designing this experiment are discussed in this section. In particular, this chapter describes the experiment layout, and the graph generation. Some of the topics include the tools used for random graph generations, and the types of graphs used in this experiment.

Experiment Layout

This section describes the reasoning behind the design of this experiment. This includes the reasoning behind selecting the graph algorithms, computing the execution times, modifying certain graph attributes, etc.

The Datalog engines were chosen based on language features that each of the Datalog engines must have. The engine's compatibility with the test equipment was taken into consideration as well. In this experiment, all of the Datalog engines must possess two important Datalog functionalities, aggregate functions and comparison operators. Both of these language attributes are used in the experiment. This experiment tests PA-Datalog, Soufflé, and Datalog Educational System (DES), all of which have these capabilities. Having consistent language features allows the rules to be near identical amongst the engines. Therefore, the results focus on a correlation between the speed of the Datalog engines and the problem size. In addition to the language features, all of the engines must run on a Linux operating system.

The experiment tests the Datalog engines on directed weighted graphs to allow for a scalable problem set and for an additional algorithm to be tested. Python and a software package, NetworkX, can generate a random graph of any size. The Python script converts the graph into facts for the Datalog engines to execute their queries. A directed graph is used because the experiment utilizes extra data from an additional graph algorithm, the strongly connected components algorithm, which only works on directed graphs.

In order to test the execution times of the Datalog engines, the Python script generates graphs with different attributes. The two main characteristics of the graph are the number of nodes and the graph density. The density is calculated using the equation in Figure 6, E is number of edges, and V is the number of nodes [3]. Table 1 and Table 2 note the graph attributes used in the experiment. Notice the shortest path algorithm requires graphs with fewer nodes due to the higher computational complexity of the algorithm. For each of the 20 characteristics in Table 1 and Table 2, three different random graphs are generated. Therefore, for each graph type and algorithm combination, there are three execution times. These execution times are averaged for the result.

$$D = \frac{|E|}{|V|(|V|-1)}$$

Figure 6. Density Equation for a Graph

Table 1. Graph Characteristics for Weakly Connected Components, Strongly Connected Components, and Connectivity

Nodes	Density (%)	Edges
50	0.5	13
50	25	612
50	50	1225
50	75	1838
100	0.5	50
100	35	2475
100	50	4950
100	75	7425
200	0.5	199
200	25	9950

Table 2. Graph Characteristics for Shortest Path

Nodes	Density (%)	Edges
10	25	23
10	50	45
10	75	68
20	25	95
20	50	190
20	75	285
30	25	218
30	50	435
40	25	390
40	50	780

To attain the execution times of the Datalog engines, the experiment uses tools such as the Linux command line tool "/usr/bin/time" and the timing tool on DES. For Soufflé and PA-Datalog, the command "/usr/bin/time –v" displays the "estimated wall clock time" of the command right after the output is displayed. The "estimated wall clock time" is the execution time of the query. Data collection for DES works differently. DES has its own user interface inside the command line, therefore utilizing the "/usr/bin/time –v" command would give incorrect results. To get around this, DES has its own timing option that calculates the execution time of every command run within its interpreter. The execution time displayed in the user interface is considered the execution time of the query.

Each Datalog engine has different steps to execute a query, and therefore different ways to evaluate execution time. To evaluate execution time for PA-Datalog, the times to create a database, to add rules to the database, to execute facts on the database, and to query facts on a database are summed together to get a total execution time. To evaluate the execution time for Soufflé, the time it takes to execute the 'souffle' command on the terminal is the execution time. To evaluate the execution time for DES, the time to consult a Datalog file, and the time to execute a query are summed together for the total execution time.

For this experiment, the Datalog engines run on a Dell Latitude E6420 with an Intel core i5 2.5GHz processor, 4 GB of RAM, and 320GB hard drive. The operating system is a Lubuntu distribution that utilizes Ubuntu 17.04.

Sample Data Generation

To generate graphs, the experiment utilizes the programming language, Python, and a library called NetworkX to generate a random graph with the appropriate number of nodes and edges. Then, the Python script creates the EDB and/or rules files for each of the Datalog engines in the experiment.

Python uses NetworkX and its built-in random graph generator to create the Datalog files. NetworkX is a language software package that is used to model and manipulate complex networks [11]. It comes with a random graph generator that takes the number of nodes, the number of edges, and the edge type, directed or undirected, and generates one graph at random from all of the possible graphs. The graph is stored in Python as a directed simple graph. Then, Python extracts the edge facts and node facts and inserts them into Datalog files.

The Python script generates nine different files. One file is generated for PA-Datalog. This file includes all of EDB. Four files are generated for Soufflé. The Python script generates a file for each of the graph algorithms. Every file includes a list of the EDB along with the rules for one graph algorithm. The script generates four files for the DES engine as well. Much like Soufflé, there is a file for each graph algorithm. Each file contains all of the EDB and the rules for one graph algorithm.

The Python script is listed in Appendix H.

Chapter 4

Results

Four graphs in this section represent the performance of the Datalog engines. Each graph indicates the performance of the Datalog engines when using one graph algorithm. In each of the graphs, the x-axis indicates the number nodes in the graph and the density of the graph. The y-axis represents the average run time for three random graphs. Some data points that made the graph unreadable are removed. Either the data point exceeded the 30-minute time limit, or the data point's value detracts from the graph's readability. All the data for the experiment is found in the appropriate appendices. Appendix A holds the data for the connectivity algorithm. Appendix B holds the data for the strongly connected components algorithm. Appendix C holds the data for the weakly connected components algorithm. Finally, Appendix D holds the data for the shortest path algorithm.

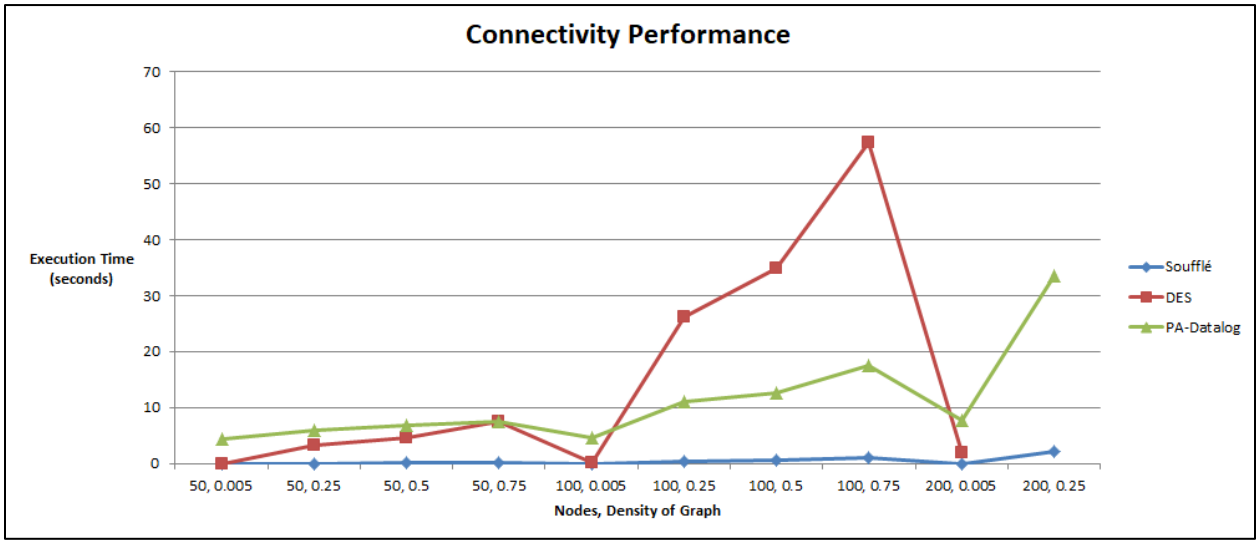

Figure 7. Connectivity Performance Graph

Figure 7 shows the performance of the Datalog engines for the connectivity rules. For this algorithm, Soufflé performed the best. It never exceeded 10 seconds and the execution times

do not drastically increase for graphs with high densities and high node numbers. DES performs well for low node counts and low densities, beating PA-Datalog on multiple data points, but the execution time increases exponentially at higher densities. PA-Datalog shows more consistency than DES, but for low density and low node numbers PA-Datalog performs the worst. This trend continues in most of the results. PA-Datalog, most likely, performs worse on these data points because it always has to execute multiple commands to create a database, to add the rules to database, to execute the facts on the database, and to print the query to the terminal. All of these commands add to the execution time, and no other engine explicitly requires these steps.

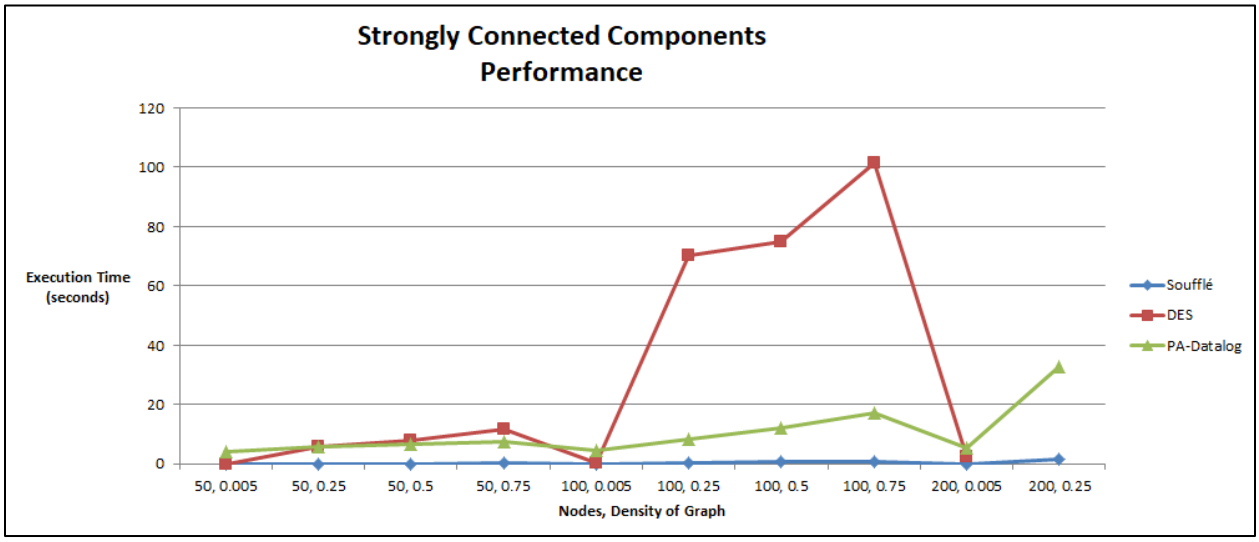

Figure 8. Strongly Connected Components Performance Graph

Figure 8 shows the results of the strongly connected components algorithm. Again, Soufflé performs the best of the three engines tested. It never exceeds 10 seconds and shows no drastic performance increase at high node counts and at high graph densities. PA-Datalog outperforms DES for most graphs except for combination of low node counts and low densities. For the higher node counts and higher densities, PA-Datalog's execution time does not increase as substantially as DES.

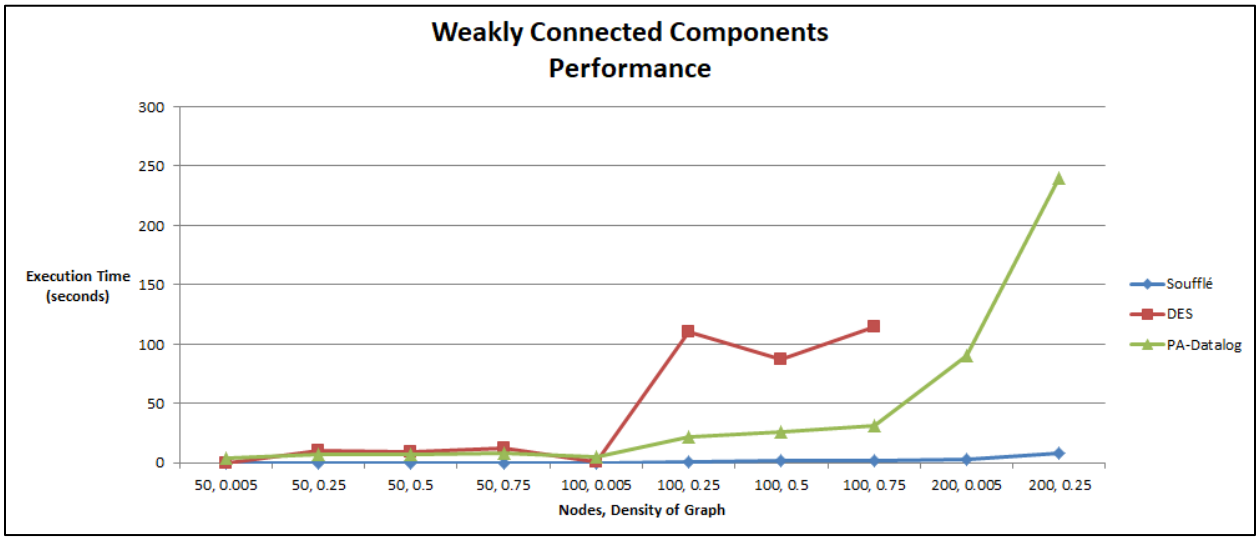

Figure 9. Weakly Connected Components Performance Graph

Figure 9 shows the performance of the three different Datalog engines when executing the weakly connected components rules. Much like the previous algorithms, Soufflé performs the best. In comparison to the other Datalog engines, the execution times do not increase much due to high node counts and high graph densities. DES and PA-Datalog perform similarly for graphs with 50 nodes and for the graph of 100 nodes and 0.5% graph density. Yet for all other graphs, PA-Datalog performs significantly better than DES despite showing an exponential increase in execution times.

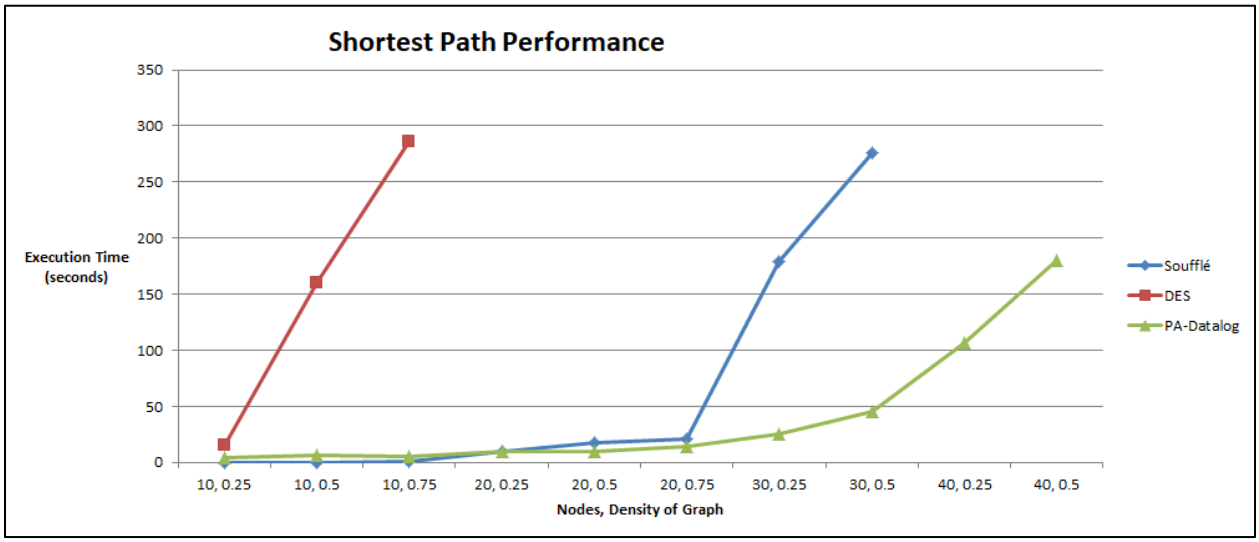

Figure 10. Shortest Path Performance Graph

Figure 10 shows the performance of the three Datalog engines when executing the shortest path algorithms. Unlike the previous test graphs, the Datalog engines ran on smaller graphs with only 10, 20, and 30 nodes. It is necessary to use a smaller dataset due to the computational complexity of the algorithm. For the shortest path algorithm, PA-Datalog shows the best performance for high node and high density graphs. Otherwise, Soufflé performed better than PA-Datalog for 10-node graphs, and performed comparably to PA-Datalog for 20-node graphs. For 30-node graphs and 40-node graphs, Soufflé showed a dramatic increase in execution time while PA-Datalog showed a more gradual increase in execution time. DES exceeded the 30-minute time limit for all, but the 10-node graphs.

Chapter 5

Summary and Conclusion

Using 4 graph algorithms and 60 randomly generated graphs, the experiment found the strongest performers for each graph algorithm. Soufflé is the strongest performer for the connectivity algorithm, strongly connected components algorithm, and the weakly connected components algorithm. For the shortest path algorithm, PA-Datalog has the lowest execution times. This shows that no Datalog engine performs best with increasing data sizes. Although, choosing the best performer can make a remarkable difference in execution times. Based on the lack of a clear best performer, there remains more work to be done in performance analysis in Datalog engines.

Due to the inconclusive results, more time needs to be spent examining why the shortest path algorithm led to different results than all of the other algorithms. Other future work includes examining execution times of Datalog engines when using high performance computing. In addition, research could examine the performance advantages of aggregate functions in different Datalog engines. Finally, some Datalog engines provide performance optimizations, such as parallel execution and C++ program generation, which may drastically affect the performance of certain queries. Future work can evaluate the advantages of using these performance optimizations.

Appendix A

Performance Results for Connectivity Algorithm

The following tables represent the performance of the Soufflé, PA-Datalog, and DES engines for the connectivity algorithm.

Table 3. Connectivity Results for 50 Nodes and 0.005 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.01	0.08	4.46
Run 2:	0.02	0.083	4.47
Run 3:	0.01	0.08	4.45
Average Run Time:	0.01333	0.081	4.46

Table 4. Connectivity Results for 50 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.07	3.321	5.99
Run 2:	0.08	3.335	6.05
Run 3:	0.08	3.39	5.85
Average Run Time:	0.07667	3.34867	5.96333

Table 5. Connectivity Results for 50 Nodes and 0.5 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.13	4.639	6.99
Run 2:	0.13	4.757	6.7
Run 3:	0.13	4.677	6.85
Average Run Time:	0.13	4.691	6.84667

Table 6. Connectivity Results for 50 Nodes and 0.75 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.19	7.476	7.63
Run 2:	0.19	7.601	7.59
Run 3:	0.18	7.566	7.66
Average Run Time:	0.18667	7.54767	7.62667

Table 7. Connectivity Results for 100 Nodes and 0.005 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.02	0.165	4.79
Run 2:	0.02	0.172	4.69
Run 3:	0.02	0.173	4.71
Average Run Time:	0.02	0.17	4.73

Table 8. Connectivity Results for 100 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.65	26.39	15.58
Run 2:	0.37	26.112	8.73
Run 3:	0.37	25.96	8.79
Average Run Time:	0.4633	26. 154	11.0333

Table 9. Connectivity Results for 100 Nodes and 0.5 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.73	34.812	13.02
Run 2:	0.72	35.189	12.74
Run 3:	0.72	34.768	12.22
Average Run Time:	0.72333	34.923	12.66

Table 10. Connectivity Results for 100 Nodes and 0.75 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.98	58.087	17.95
Run 2:	0.99	56.878	17.58
Run 3:	1.03	57.326	17.5
Average Run Time:	1.0	57.4303	17.6767

Table 11. Connectivity Results for 200 Nodes and 0.005 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.04	1.13	5.47
Run 2:	0.05	3.921	5.48
Run 3:	0.04	1.193	12.22
Average Run Time:	0.04333	2.08133	7.72333

Table 12. Connectivity Results for 200 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	2.26	161.6	34.06
Run 2:	2.28	163.637	33.75
Run 3:	2.26	160.637	33.14
Average Run Time:	2.2667	162.038	33.65

Appendix B

Performance Results for Strongly Connected Components Algorithm

The following tables represent the performance of the Soufflé, PA-Datalog, and DES engines for the strongly connected components algorithm.

Table 13. Strongly Connected Components Results for 50 Nodes and 0.005 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.02	0.099	4.27
Run 2:	0.02	0.109	4.27
Run 3:	0.01	0.1	4.25
Average Run Time:	0.01667	0.10267	4.26333

Table 14. Strongly Connected Components Results for 50 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.07	5.579	5.78
Run 2:	0.07	5.56	5.67
Run 3:	0.07	5.714	5.76
Average Run Time:	0.07	5.61767	5.73667

Table 15. Strongly Connected Components Results for 50 Nodes and 0.5 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.12	7.685	6.71
Run 2:	0.13	7.781	6.67
Run 3:	0.13	7.719	6.8
Average Run Time:	0.12667	7.72833	6.72667

Table 16. Strongly Connected Components Results for 50 Nodes and 0.75 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.18	11.375	7.44
Run 2:	0.18	11.682	7.28
Run 3:	0.18	11.618	7.37
Average Run Time:	0.18	11.5583	7.36333

Table 17. Strongly Connected Components Results for 100 Nodes and 0.005 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.02	0.208	4.46
Run 2:	0.03	0.214	4.52
Run 3:	0.02	0.215	4.5
Average Run Time:	0.02333	0.21233	4.49333

Table 18. Strongly Connected Components Results for 100 Nodes and 0. 25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.31	69.32	8.34
Run 2:	0.31	71.608	8.2
Run 3:	0.3	70.005	8.16
Average Run Time:	0.30667	70.311	8.2333

Table 19. Strongly Connected Components Results for 100 Nodes and 0.5 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.63	74.196	12.29
Run 2:	0.64	75.644	12.19
Run 3:	0.64	72.676	12.35
Average Run Time:	0.63667	74.172	12.2767

Table 20. Strongly Connected Components Results for 100 Nodes and 0.75 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.91	101.57	17.13
Run 2:	0.91	101.596	17.6
Run 3:	0.92	103.148	17.41
Average Run Time:	0.91333	102.105	17.38

Table 21. Strongly Connected Components Results for 200 Nodes and 0.005 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.04	1.46	5.51
Run 2:	0.05	4.484	5.26
Run 3:	0.03	1.553	5.35
Average Run Time:	0.04	2.499	5.37333

Table 22. Strongly Connected Components Results for 200 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	1.75	689.08	33.51
Run 2:	1.74	691.765	32.8
Run 3:	1.74	668.677	31.99
Average Run Time:	1.74333	663.174	32.7667

Appendix C

Performance Results for Weakly Connected Components Algorithm

The following tables represent the performance of the Soufflé, PA-Datalog, and DES engines for the weakly connected components algorithm.

Table 23. Weakly Connected Components Results for 50 Nodes and 0.005 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.02	0.136	4.29
Run 2:	0.02	0.139	4.3
Run 3:	0.02	0.133	4.27
Average Run Time:	0.02	0.136	4.28667

Table 24. Weakly Connected Components Results for 50 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.17	10.488	6.65
Run 2:	0.17	10.482	6.7
Run 3:	0.17	10.642	6.79
Average Run Time:	0.17	10.5373	6.71333

Table 25. Weakly Connected Components Results for 50 Nodes and 0.5 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.22	9.14	7.49
Run 2:	0.21	9.19	7.51
Run 3:	0.22	9.138	7.91
Average Run Time:	0.21667	9.156	7.63667

Table 26. Weakly Connected Components Results for 50 Nodes and 0.75 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.27	11.927	8.47
Run 2:	0.27	12.23	8.41
Run 3:	0.27	12.25	8.16
Average Run Time:	0.27	12.1357	8.34667

Table 27. Weakly Connected Components Results for 100 Nodes and 0.005 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.04	1.71	4.71
Run 2:	0.02	0.647	4.65
Run 3:	0.04	1.154	4.48
Average Run Time:	0.03333	1.17033	4.61333

Table 28. Weakly Connected Components Results for 100 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	1.06	110.88	21.68
Run 2:	1.09	108.51	21.75
Run 3:	1.11	111.16	21.9
Average Run Time:	1.08667	110.183	21.7767

Table 29. Weakly Connected Components Results for 100 Nodes and 0.5 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	1.39	87.285	26.03
Run 2:	1.4	86.697	25.63
Run 3:	1.4	86.036	26.03
Average Run Time:	1.39667	86.6727	25.8967

Table 30. Weakly Connected Components Results for 100 Nodes and 0.75 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	1.67	114.407	30.46
Run 2:	1.67	113.782	31.35
Run 3:	1.67	113.113	30.63
Average Run Time:	1.67	113.767	30.8133

Table 31. Weakly Connected Components Results for 200 Nodes and 0.005 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	3.59		102.84
Run 2:	2.62		69.96
Run 3:	3.45		98.55
Average Run Time:	3.22	EXCEEDED LIMIT	90.45

Table 32. Weakly Connected Components Results for 200 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	7.93	861.098	240.97
Run 2:	8.72	864.946	239.37
Run 3:	7.83	906.455	238.3
Average Run Time:	8.16	877.5	239.547

Appendix D

Performance Results for Shortest Path Algorithm

The following tables represent the performance of the Soufflé, PA-Datalog, and DES engines for the shortest path algorithm.

Table 33. Shortest Path Results for 10 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.01	32.818	4.62
Run 2:	0.05	10.842	4.58
Run 3:	0.04	3.667	4.62
Average Run Time:	0.03333	15.7757	4.60667

Table 34. Shortest Path Results for 10 Nodes and 0.5 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.26	178.367	4.84
Run 2:	0.42	151.542	11.31
Run 3:	0.23	149.596	4.75
Average Run Time:	0.30333	159.835	6.96667

Table 35. Shortest Path Results for 10 Nodes and 0.75 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	0.4	304.294	4.97
Run 2:	0.41	300.093	4.85
Run 3:	0.38	255.03	4.83
Average Run Time:	0.39667	286.472	4.88333

Table 36. Shortest Path Results for 20 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	9.63		14.01
Run 2:	9.45		7.35
Run 3:	10.36		7.4
Average Run Time:	9.81333	EXCEEDED LIMIT	9.58667

Table 37. Shortest Path Results for 20 Nodes and 0.5 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	18.06		10.12
Run 2:	16.37		9.81
Run 3:	16.82		9.94
Average Run Time:	17.0833	EXCEEDED LIMIT	9.95667

Table 38. Shortest Path Results for 20 Nodes and 0.75 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	21.45		12.22
Run 2:	20.41		12.04
Run 3:	21.55		19.3
Average Run Time:	21.1367	EXCEEDED LIMIT	14.52

Table 39. Shortest Path Results for 30 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	185.87		25.81
Run 2:	170.9		25.5
Run 3:	180.41		25.52
Average Run Time:	179.06	EXCEEDED LIMIT	25.61

Table 40. Shortest Path Results for 30 Nodes and 0.5 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:	268.74		43.23
Run 2:	282.13		43.21
Run 3:	275.85		46.69
Average Run Time:	275.573	EXCEEDED LIMIT	45.3767

Table 41. Shortest Path Results for 40 Nodes and 0.25 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:			101.73
Run 2:			111.8
Run 3:			107.95
Average Run Time:	EXCEEDED LIMIT	EXCEEDED LIMIT	107.16

Table 42. Shortest Path Results for 40 Nodes and 0.5 Graph Density

	Soufflé	DES	PA-Datalog
Run 1:			178.82
Run 2:			185.46
Run 3:			177.65
Average Run Time:	EXCEEDED LIMIT	EXCEEDED LIMIT	180.643

Appendix E

Code for PA-Datalog

This section displays the rules used in the testing of the PA-Datalog engine. The first code is for **connectivity**.

```
// PA-DATALOG

// All possible facts
node(x)             -> int[64](x).
edge(x, y, weight) -> int[64](x), int[64](y), int[64](weight).
path(x, y)          -> int[64](x), int[64](y).
num_nodes(num)      -> int[64](num).

// the derivation for the path facts
path(x, z)  <- edge(x, z, _), node(x), node(z).
path(x, z)  <- path(x, y), edge(y, z, _), node(x), node(y), node(z).
```

This code is for **strongly connected components**.

```
// PA-DATALOG

// all possible facts
node(x)                -> int[64](x).
edge(x, y, weight)     -> int[64](x), int[64](y), int[64](weight).
path(x, y)             -> int[64](x), int[64](y).
scc(x, y)              -> int[64](x), int[64](y).
scc_g_init(x, s)       -> int[64](x), int[64](s).
scc_g(x, s)            -> int[64](x), int[64](s).
num_nodes(num)         -> int[64](num).

// rule used to establish which nodes are connected
path(x, z)  <- edge(x, z, _), node(x), node(z).
path(x, z)  <- path(x, y), edge(y, z, _), node(x), node(y), node(z).

// strongly connected nodes has path to one another in both
// directions
// the smaller number is kept on the left
scc(x, y) <- path(x, y), path(y, x), x < y.

// strongly connected graph initialization rule
// read it as follows
scc_g_init(x, s) <- scc(x, _), !scc(_, x), s = x.

// the strongly_connected_graph identifies each graph by its
// lowest node value
```

```
// the left value will be the identifier of the strongly connected
graph
// the right value will be the member of the strongly connected graph
// each strongly connected graph is identified by the value of
// smallest member value
scc_g(x, s) <- scc_g_init(x, s).
scc_g(x, y) <- scc_g_init(x, s), scc(s, y).
```

This code is for **weakly connected components**.

```
// PA-DATALOG

// instantiate all of the possible facts
node(x)              -> int[64](x).
edge(x, y, weight)   -> int[64](x), int[64](y), int[64](weight).
path(x, y)           -> int[64](x), int[64](y).
wcc(nd1, nd2)        -> int[64](nd1),  int[64](nd2).
wcc_g[nd1] = nd2     -> int[64](nd1),  int[64](nd2).
num_nodes(num)       -> int[64](num).

// connectivity from one node to another
path(x, z)  <- edge(x, z, _), node(x), node(z).
path(x, z)  <- path(x, y), edge(y, z, _), node(x), node(y), node(z).


// if all directed edges were replaced by undirected edges,
// any of the edges that are connected via an edge are considered
// weakly connected.
// for wcc rule the smaller number should be on the left
wcc(nd, nd)   <- node(nd).
wcc(nd1, nd2) <- path(nd1, nd2), nd1 < nd2.
wcc(nd1, nd2) <- path(nd2, nd1), nd1 < nd2.

// a weakly connected components derived from the transitive property
wcc(nd1, nd3) <- wcc(nd1, nd2), wcc(nd2, nd3).
wcc(nd1, nd3) <- wcc(nd1, nd2), wcc(nd3, nd2).

// to find which nodes reside in a common weakly connected graph
wcc_g[nd2] = nd  <- agg<<nd = min(nd1)>> wcc(nd1, nd2).
```

The last code segment is the algorithm for **shortest path**.

```
// PA-DATALOG

// Instantiation of all possible facts
node(x)                                -> int[64](x).
edge(x, y, weight)                     -> int[64](x), int[64](y),
int[64](weight).
```

```
min_path[src, dest] = dist            -> int[64](dest), int[64](dist),
int[64](src).
path_dis(src, dest, dist, rnd)        -> int[64](src),  int[64](dest),
int[64](dist), int[64](rnd).
num_nodes(num)                        -> int[64](num).


// the path_dis fact tells the distance from the src node to the
// destination node
// the round number is used to end the algorithm
// path_dis(source, destination, distance, round)
path_dis(n,n,0,0)            <- node(n).

// this algorithm is similar to Bellman-Ford algorithm for shortest
path
// all possible shortest paths are instantiated and stored as path_dis
// facts
path_dis(src,dest,dist,rnd1) <- path_dis(src, n, dist1, rnd),
edge(n,dest,dist2), num_nodes(t), dist=dist1+dist2, rnd1=rnd+1, rnd<t.

// min_path uses an aggregate function to find the shortest path for
// every source and destination node found in path_dis.
min_path[src, dest] = d            <- agg<<d = min(dist)>>
path_dis(src, dest, dist, _).
```

Appendix F

Code for Soufflé

This section displays the rules used in the testing of the Soufflé engine. The first code is for **connectivity**.

```
// Types created by the user
.number_type nodes

// Declared Facts
.decl node(n:nodes)
.decl num_nodes(n:nodes)
.decl edge(n:nodes, m:nodes, wt:number)
.decl path(n:nodes, m:nodes)

// The Facts that will outputted at the end of the query
.output path(IO=stdout)

// Rules for computing paths
path(X, Z)  :- edge(X, Z, _), node(X), node(Z).
path(X, Z)  :- path(X, Y), edge(Y, Z, _), node(X), node(Y), node(Z).
```

This code is for **strongly connected components**.

```
// Types created by the user
.number_type nodes

// Declared Facts
.decl node(n:nodes)
.decl num_nodes(n:nodes)
.decl edge(n:nodes, m:nodes, wt:number)
.decl path(n:nodes, m:nodes)
.decl scc(n:nodes, m:nodes)
.decl scc_g(n:nodes, m:nodes)
.decl scc_g_init(n:nodes, m:nodes)

// The Facts that will outputted at the end of the query
.output scc_g(IO=stdout)

// create all path
path(X, Z)  :- edge(X, Z, _), node(X), node(Z).
path(X, Z)  :- path(X, Y), edge(Y, Z, _), node(X), node(Y), node(Z).

// identify strongly connected components
scc(X, Y) :- path(X, Y), path(Y, X), X < Y.
```

```
// initially strongly connected subgraphs
scc_g_init(X, S) :- scc(X, _), !scc(_, X), S = X.

// calculate components of the subgraph
scc_g(X, S) :- scc_g_init(X, S).
scc_g(X, Y)  :- scc_g_init(X, S), scc(S, Y).
```

This code is for **weakly connected components**.

```
// Types created by the user
.number_type nodes

// Declared Facts
.decl node(n:nodes)
.decl num_nodes(n:nodes)
.decl edge(n:nodes, m:nodes, wt:number)
.decl path(n:nodes, m:nodes)
.decl wcc(n:nodes, m:nodes)
.decl wcc_g(n:nodes, m:nodes)

// The Facts that will outputted at the end of the query
.output wcc_g(IO=stdout)

// create all of the paths
path(X, Z)  :- edge(X, Z, _), node(X), node(Z).
path(X, Z)  :- path(X, Y), edge(Y, Z, _), node(X), node(Y),
node(Z).

// find all weakly connected components
wcc(nd, nd)   :- node(nd).
wcc(nd1, nd2) :- path(nd1, nd2), nd1 < nd2.
wcc(nd1, nd2) :- path(nd2, nd1), nd1 < nd2.
wcc(nd1, nd3) :- wcc(nd1, nd2), wcc(nd2, nd3).
wcc(nd1, nd3) :- wcc(nd1, nd2), wcc(nd3, nd2).

// using the weakly connected components find the subgraphs
wcc_g(nd2, nd)  :- wcc(_, nd2), nd = min nd1 : wcc(nd1, nd2).
```

The last code segment is the algorithm for **shortest path**.

```
// Types created by the user
.number_type nodes

// Declared Facts
.decl node(n:nodes)
.decl num_nodes(n:nodes)
.decl edge(n:nodes, m:nodes, wt:number)
.decl path_dis(src:nodes, dest:nodes, dist:number, rnd:number)
```

```
.decl min_path(src:nodes, dest:nodes, dist:number)

// The Facts that will outputted at the end of the query
.output min_path(IO=stdout)

// find all of the distances between two nodes
path_dis(n, n, 0, 0)  :- node(n).
path_dis(src, dest, dist1 + dist2, rnd + 1) :- path_dis(src, n, dist1,
rnd), edge(n, dest, dist2), num_nodes(t), rnd<t.

// calculate the minimum path
min_path(src, dest, d)                    :- path_dis(src, dest, _,
_), d = min dist : path_dis(src, dest, dist, _).
```

Appendix G

Code for Datalog Educational System

This section displays the rules used in the testing of the DES engine. The first code is for **connectivity**.

```
// a path is an edge between two nodes
path(X, Z)  :- edge(X, Z, _), node(X), node(Z).
// a path is a path with an edge to another node.
path(X, Z)  :- path(X, Y), edge(Y, Z, _), node(X), node(Y), node(Z).
```

This code is for **strongly connected components**.

```
// instantiate all of the paths in the graph
path(X, Z)  :- edge(X, Z, _), node(X), node(Z).
path(X, Z)  :- path(X, Y), edge(Y, Z, _), node(X), node(Y), node(Z).

// instantiate all of the strongly connected components in the graph
scc(X, Y) :- path(X, Y), path(Y, X), X < Y.

// initialize the strongly connected subgraphs
scc_g_init(X, S) :- scc(X, _), not scc(_, X), S = X.

// define the strongly connected subgraphs based on the initial graphs
scc_g(X, S) :- scc_g_init(X, S).
scc_g(X, Y) :- scc_g_init(X, S), scc(S, Y).
```

This code is for **weakly connected components**.

```
// instantiate all of the paths in the graph
path(X, Z)  :- edge(X, Z, _), node(X), node(Z).
path(X, Z)  :- path(X, Y), edge(Y, Z, _), node(X), node(Y), node(Z).

// find all weakly connected components
wcc(N, N)   :- node(N).
wcc(N, M) :- path(N, M), N < M.
wcc(N, M) :- path(M, N), N < M.
wcc(N, O) :- wcc(N, M), wcc(M, O).
wcc(N, O) :- wcc(N, M), wcc(O, M).

// based on the wccs create the weakly connected subgraphs
wcc_g(M, S)  :- min(wcc(N, M), N, S).
```

The last code segment is the algorithm for **shortest path**.

```
// initialize the path distances
```

```
path_dis(N,N,Z,Z)   :- node(N), Z = 0.
// determine all of the distances between nodes in the graph
path_dis(S, D, DI, R1) :- path_dis(S, N, D1, R), edge(N, D, D2),
num_nodes(T), R<T, DI = D1 + D2, R1 = R + 1.

// use the min aggregate to find the shortest path between nodes
min_path(S, DE, M)             :- min(path_dis(S, DE, D,_), D, M).
```

Appendix H

Python Script for Random Graph Generation and File Creation

This appendix holds the Python script that creates a random graph and outputs the contents of the graph to multiple Datalog files.

```
import networkx as nx
import random

file_pa_datalog       = open("PA-DATALOG/pa_datalog_facts.logic", 'w')
file_souffle_conn     =
open("Souffle/Datalog_Files/souffle_logic_path.dl", "w")
file_souffle_scc      =
open("Souffle/Datalog_Files/souffle_logic_scc.dl", "w")
file_souffle_wcc      =
open("Souffle/Datalog_Files/souffle_logic_wcc.dl", "w")
file_souffle_min_path =
open("Souffle/Datalog_Files/souffle_logic_min_path.dl", "w")
file_des_conn         = open("DES/datalog_files/des_logic_path.dl",
"w")
file_des_scc          = open("DES/datalog_files/des_logic_scc.dl",
"w")
file_des_wcc          = open("DES/datalog_files/des_logic_wcc.dl",
"w")
file_des_min_path     =
open("DES/datalog_files/des_logic_min_path.dl", "w")

NUM_NODES = 200
NUM_EDGES = 199

G = nx.DiGraph()

G = nx.gnm_random_graph(NUM_NODES, NUM_EDGES, directed = True)

nodes_list = nx.nodes(G)

edge_list = nx.to_edgelist(G)



file_pa_datalog.write("+num_nodes({0}).\n".format(NUM_NODES))

file_souffle_conn.write("// Types created by the user\n")
file_souffle_conn.write(".number_type nodes\n")
file_souffle_conn.write("\n// Declared Facts\n")
file_souffle_conn.write(".decl node(n:nodes)\n")
```

```
file_souffle_conn.write(".decl num_nodes(n:nodes)\n")
file_souffle_conn.write(".decl edge(n:nodes, m:nodes, wt:number)\n")
file_souffle_conn.write(".decl path(n:nodes, m:nodes)\n")
file_souffle_conn.write("\n// The Facts that will outputted at the end
of the query\n")
file_souffle_conn.write(".output path(IO=stdout)\n")
file_souffle_conn.write("\npath(X, Z)  :- edge(X, Z, _), node(X),
node(Z).\n")
file_souffle_conn.write("path(X, Z)  :- path(X, Y), edge(Y, Z, _),
node(X), node(Y), node(Z).\n")
file_souffle_conn.write("num_nodes({0}).\n".format(NUM_NODES))

file_souffle_scc.write("// Types created by the user\n")
file_souffle_scc.write(".number_type nodes\n")
file_souffle_scc.write("\n// Declared Facts\n")
file_souffle_scc.write(".decl node(n:nodes)\n")
file_souffle_scc.write(".decl num_nodes(n:nodes)\n")
file_souffle_scc.write(".decl edge(n:nodes, m:nodes, wt:number)\n")
file_souffle_scc.write(".decl path(n:nodes, m:nodes)\n")
file_souffle_scc.write(".decl scc(n:nodes, m:nodes)\n")
file_souffle_scc.write(".decl scc_g(n:nodes, m:nodes)\n")
file_souffle_scc.write(".decl scc_g_init(n:nodes, m:nodes)\n")
file_souffle_scc.write("\n// The Facts that will outputted at the end
of the query\n")
file_souffle_scc.write(".output scc_g(IO=stdout)\n")
file_souffle_scc.write("\npath(X, Z)  :- edge(X, Z, _), node(X),
node(Z).\n")
file_souffle_scc.write("path(X, Z)  :- path(X, Y), edge(Y, Z, _),
node(X), node(Y), node(Z).\n")
file_souffle_scc.write("\nscc(X, Y) :- path(X, Y), path(Y, X), X <
Y.\n")
file_souffle_scc.write("\nscc_g_init(X, S) :- scc(X, _), !scc(_, X), S
= X.\n")
file_souffle_scc.write("\nscc_g(X, S) :- scc_g_init(X, S).\n")
file_souffle_scc.write("scc_g(X, Y)  :- scc_g_init(X, S), scc(S,
Y).\n\n")
file_souffle_scc.write("num_nodes({0}).\n".format(NUM_NODES))

file_souffle_wcc.write("// Types created by the user\n")
file_souffle_wcc.write(".number_type nodes\n")
file_souffle_wcc.write("\n// Declared Facts\n")
file_souffle_wcc.write(".decl node(n:nodes)\n")
file_souffle_wcc.write(".decl num_nodes(n:nodes)\n")
file_souffle_wcc.write(".decl edge(n:nodes, m:nodes, wt:number)\n")
file_souffle_wcc.write(".decl path(n:nodes, m:nodes)\n")
file_souffle_wcc.write(".decl wcc(n:nodes, m:nodes)\n")
file_souffle_wcc.write(".decl wcc_g(n:nodes, m:nodes)")
file_souffle_wcc.write("\n// The Facts that will outputted at the end
of the query\n")
file_souffle_wcc.write(".output wcc_g(IO=stdout)")
```

```
file_souffle_wcc.write("\npath(X, Z)  :- edge(X, Z, _), node(X),
node(Z).\n")
file_souffle_wcc.write("path(X, Z)  :- path(X, Y), edge(Y, Z, _),
node(X), node(Y), node(Z).\n")
file_souffle_wcc.write("wcc(nd, nd)   :- node(nd).\n")
file_souffle_wcc.write("wcc(nd1, nd2) :- path(nd1, nd2), nd1 <
nd2.\n")
file_souffle_wcc.write("wcc(nd1, nd2) :- path(nd2, nd1), nd1 <
nd2.\n")
file_souffle_wcc.write("wcc(nd1, nd3) :- wcc(nd1, nd2), wcc(nd2,
nd3).\n")
file_souffle_wcc.write("wcc(nd1, nd3) :- wcc(nd1, nd2), wcc(nd3, nd2).
\n")
file_souffle_wcc.write("wcc_g(nd2, nd)  :- wcc(_, nd2), nd = min nd1 :
wcc(nd1, nd2).\n")
file_souffle_wcc.write("num_nodes({0}).\n".format(NUM_NODES))


file_souffle_min_path.write("// Types created by the user\n")
file_souffle_min_path.write(".number_type nodes\n")
file_souffle_min_path.write("\n// Declared Facts\n")
file_souffle_min_path.write(".decl node(n:nodes)\n")
file_souffle_min_path.write(".decl num_nodes(n:nodes)\n")
file_souffle_min_path.write(".decl edge(n:nodes, m:nodes,
wt:number)\n")
file_souffle_min_path.write(".decl path_dis(src:nodes, dest:nodes,
dist:number, rnd:number)\n")
file_souffle_min_path.write(".decl min_path(src:nodes, dest:nodes,
dist:number)\n")
file_souffle_min_path.write("\n// The Facts that will outputted at the
end of the query\n")
file_souffle_min_path.write(".output min_path(IO=stdout)\n")
file_souffle_min_path.write("\npath_dis(n, n, 0, 0)  :- node(n).\n")
file_souffle_min_path.write("path_dis(src, dest, dist1 + dist2, rnd +
1) :- path_dis(src, n, dist1, rnd), edge(n, dest, dist2),
num_nodes(t), rnd<t.\n\n")
file_souffle_min_path.write("\nmin_path(src, dest, d)
:- path_dis(src, dest, _, _), d = min dist : path_dis(src, dest, dist,
_).\n\n")
file_souffle_min_path.write("num_nodes({0}).\n".format(NUM_NODES))

file_des_conn.write("path(X, Z)  :- edge(X, Z, _), node(X),
node(Z).\n")
file_des_conn.write("path(X, Z)  :- path(X, Y), edge(Y, Z, _),
node(X), node(Y), node(Z).\n\n")
file_des_conn.write("num_nodes({0}).\n".format(NUM_NODES))

file_des_scc.write("path(X, Z)  :- edge(X, Z, _), node(X),
node(Z).\n")
```

```
file_des_scc.write("path(X, Z)  :- path(X, Y), edge(Y, Z, _), node(X),
node(Y), node(Z).\n\n")
file_des_scc.write("scc(X, Y) :- path(X, Y), path(Y, X), X < Y.\n\n")
file_des_scc.write("scc_g_init(X, S) :- scc(X, _), not scc(_, X), S =
X.\n\n")
file_des_scc.write("scc_g(X, S) :- scc_g_init(X, S).\n")
file_des_scc.write("scc_g(X, Y) :- scc_g_init(X, S), scc(S, Y).\n\n")
file_des_scc.write("num_nodes({0}).\n".format(NUM_NODES))

file_des_wcc.write("path(X, Z)  :- edge(X, Z, _), node(X),
node(Z).\n")
file_des_wcc.write("path(X, Z)  :- path(X, Y), edge(Y, Z, _), node(X),
node(Y), node(Z).\n\n")
file_des_wcc.write("wcc(N, N)   :- node(N).\n")
file_des_wcc.write("wcc(N, M) :- path(N, M), N < M.\n")
file_des_wcc.write("wcc(N, M) :- path(M, N), N < M.\n")
file_des_wcc.write("wcc(N, O) :- wcc(N, M), wcc(M, O).\n")
file_des_wcc.write("wcc(N, O) :- wcc(N, M), wcc(O, M). \n")
file_des_wcc.write("wcc_g(M, S)  :- min(wcc(N, M), N, S).\n")
file_des_wcc.write("num_nodes({0}).\n".format(NUM_NODES))


file_des_min_path.write("path_dis(N,N,Z,Z)   :- node(N), Z = 0.\n")
file_des_min_path.write("path_dis(S, D, DI, R1) :- path_dis(S, N, D1,
R), edge(N, D, D2), num_nodes(T), R<T, DI = D1 + D2, R1 = R + 1.\n\n")
file_des_min_path.write("min_path(S, DE, M)             :-
min(path_dis(S, DE, D,_), D, M).\n\n")
file_des_min_path.write("num_nodes({0}).\n".format(NUM_NODES))


for node in nodes_list:

    string_to_write   = "+node({0}).\n".format(node)
    string_to_write_2 = "node({0}).\n".format(node)

    file_pa_datalog.write(string_to_write)

    file_des_conn.write(string_to_write_2)
    file_des_wcc.write(string_to_write_2)
    file_des_scc.write(string_to_write_2)
    file_des_min_path.write(string_to_write_2)

    file_souffle_conn.write(string_to_write_2)
    file_souffle_wcc.write(string_to_write_2)
    file_souffle_scc.write(string_to_write_2)
    file_souffle_min_path.write(string_to_write_2)

for edge in edge_list:

    edge_weight = random.randrange(1, 11)
```

```
    edge_string = "+edge({0}, {1}, {2}).\n".format(edge[0], edge[1],
edge_weight)

    file_pa_datalog.write(edge_string)

    edge_string2 = "edge({0}, {1}, {2}).\n".format(edge[0], edge[1],
edge_weight)

    file_souffle_conn.write(edge_string2)
    file_souffle_scc.write(edge_string2)
    file_souffle_wcc.write(edge_string2)
    file_souffle_min_path.write(edge_string2)

    file_des_conn.write(edge_string2)
    file_des_scc.write(edge_string2)
    file_des_wcc.write(edge_string2)
    file_des_min_path.write(edge_string2)

file_pa_datalog.close()

file_souffle_conn.close()
file_souffle_wcc.close()
file_souffle_scc.close()
file_souffle_min_path.close()

file_des_conn.close()
file_des_wcc.close()
file_des_scc.close()
file_des_min_path.close()
```

BIBLIOGRAPHY

[1] Bravenboer, Martin, and Yannia Smaragdakis. “Strictly Declarative Specification of Sophisticated Points-to Analyses.” *OOPSLA '09 Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pp. 243–262.

[2] Chomicki, Jan. “Data Integration: Logic Query Languages.” 22 Mar. 2018.

[3] Coleman, Thomas F., and Jorge J. More. “Estimation of Sparse Jacobian Matrices and Graph Coloring Problems.” *SIAM Journal on Numerical Analysis*, vol. 20, no. 1, pp. 187–209.

[4] *Datalog Educational System*, des.sourceforge.net/.

[5] Dubhashi, Devdatt, et al. “Fast distributed algorithms for (Weakly) connected dominating sets and linear-Size skeletons.” *Journal of Computer and System Sciences*, vol. 71, no. 4, Nov. 2005, pp. 467–479.

[6] Graham, Susan L., et al. “Gprof: A call graph execution profiler.” *SIGPLAN '82 Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pp. 120–126.

[7] Huang, Shan Shan, et al. “Datalog and Emerging Applications: An Interactive Tutorial.” *SIGMOD '11 Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pp. 1213–1216.

[8] Kumar, Ravi, et al. “The Web and Social Networks.” *Computer*, vol. 35, no. 11, 10 Dec. 2002, pp. 32–36.

[9] *LogicBlox - LogicBlox 3.10 Reference Manual*, LogicBlox, developer.logicblox.com/content/docs/core-reference/html/index.html.

[10] McCarthy, Jay. “Datalog: Deductive Database Programming.” *Racket*, docs.racket-lang.org/datalog/.

[11] *NetworkX 2.1 Documentation*, NetworkX Developers, 22 Jan. 2018, networkx.github.io/documentation/stable/.

[12] *PA-Datalog Download*, snf-705535.vm.okeanos.grnet.gr/agreement.html.

[13] Sáenz-Pérez, Fernando. “Datalog Educational System V5.0 User's Manual.” 24 Feb. 2017.

[14] Seo, Jiwon, et al. “SociaLite: An Efficient Graph Query Language Based on Datalog.” *IEEE Transactions on Knowledge and Data Engineering*, 19 Feb. 2015. IEEE Xplore, IEEE.

[15] Smaragdakis, Yannis. *Datalog 101*. bitbucket.org/yanniss/doop/src/9dc8fb66176b04df34805af0d9ee825d867ee807/docs/datalog-101.md?fileviewer=file-view-default.

[16] *Souffle*, souffle-lang.org/.

[17] Ullman, Jeffrey. “Datalog: Logical Rules, Recursion.” 22 Mar. 2018.

ACADEMIC VITA

Corey Capooci
cvc5673@psu.edu

Education

Major(s) and Minor(s): Computer Engineering
Honors: Computer Engineering

Thesis Title: Performance Analysis of Three Datalog Engines
Thesis Supervisor: Gang Tan

Work Experience

May 2017 – July 2017
Software Engineer Intern
Assisted in the transition to a new integration testing framework.
Cerner Corporation
51 Valley Stream Pkwy
Malvern, PA 19355
James Bradley and Joanna Abbruzzesi

June 2016 – August 2016
Software Engineer Intern
Developed automated test scripts in Python.
Textron Systems
124 Industry Lane
Hunt Valley, MD 21030
Steve Beck and William Langan

August 2017 – December 2017
Learning Assistant for CMPSC121
The Pennsylvania State University
Dr. Steven Shaffer

January 2018 – May 2018
Grader for CMPSC461
The Pennsylvania State University
Dr. Gang Tan

Leadership Experience

March 2017 – May 2018
Tutoring Chair
Help prepare School of Electrical Engineering and Computer Science students for classes such as CMPEN 270 and EE 210 through weekly tutoring sessions.
Eta Kappa Nu