

Shuntian Liu

Erlog: A Distributed Datalog Engine

Computer Science Tripos – Part II

Magdalene College

May 12, 2022

Declaration of originality

I, Shuntian Liu of Magdalene College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I am content for my dissertation to be made available to the students and staff of the University.

Signed: Shuntian Liu

Date: 2022-05-12

Acknowledgements

I am grateful to:

- My supervisor **Mistral Contrastin**, for his patient guidance, frequent encouragement and creative ideas.
- My Director of Studies **John Fawcett** for offering thorough and critical advice throughout the project.
- My friend **Peter Zhang** for generously providing multi-core VMs to me for the evaluation of this project, and many hours of technical discussion.
- **Andrew Rice** for introducing me to this project.
- **Anil Madhavapeddy**, **Yu He** and **Han Xuanyuan** for reading and giving feedback on this dissertation.

Proforma

Candidate Number: 2360B

Project Title: Erlog: a distributed datalog engine

Examination: Computer Science Tripos, Part II (2022)

Word Count: 11994^{*}

Lines of Code: 4525[†]

Project Originator: Mistral Contrastin

Supervisors: Mistral Contrastin and Dr A. Madhavapeddy

Original aims of the project

This project aimed to investigate to what extent datalog programs are amenable to distributed evaluation. To address this question, a distributed datalog engine was built to collect empirical data. The engine attempts to exploit the embarrassingly parallel nature of datalog program evaluation. The core technical deliverable is to have an engine that can compute the transitive closure of a medium-sized graph, which will then be evaluated in terms of its scalability and fault tolerance.

Work completed

All success criteria and selected extensions for this project have been met. This engine can indeed evaluate the transitive closure program with its correctness checked against the state-of-the-art Soufflé datalog engine.

The engine achieves strong scalability on a cluster of nodes running on a single multi-core machine connected via the loopback interface and a real cluster of distributed nodes. The engine is also tolerant against worker crashes and slowness and maintains scalability despite their presence.

Special difficulties

None.

^{*}Counted using TeXcount: <https://app.uio.no/ifi/texcount/>, retrieved on 2022-04-07

[†]Counted using cloc: <https://github.com/AlDanial/cloc>, retrieved on 2022-04-07

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Related areas	2
1.3	Dissertation structure	2
2	Preparation	3
2.1	Background knowledge	3
2.1.1	The datalog language	3
2.1.2	Fixpoint-theoretic semantics of datalog	3
2.1.3	Recursive query processing	4
2.1.4	Negation	5
2.1.5	Distributed framework	8
2.1.6	Erlang features	9
2.2	Requirements analysis	10
2.3	Software engineering techniques	11
2.3.1	Development model	11
2.3.2	Backup, version control and testing	11
2.3.3	Software license	12
2.4	Starting point	12
2.5	Summary	12
3	Implementation	13
3.1	Overview	13
3.2	Erlog syntax	14
3.3	Single node evaluation	14
3.3.1	Relational Algebra (RA)	14
3.3.2	Database operation primitives	15
3.3.3	Implement datalog as Database operations	17
3.3.4	Semi-naïve evaluation strategy	17
3.3.5	Negation	17
3.4	Distributed evaluation	18
3.4.1	Execution overview	19
3.4.2	Distributed RA operations and simple distributed join	19
3.4.3	Stratification and parallel execution	20
3.4.4	Coordinator jobs	21
3.4.5	Worker jobs	26
3.5	Implementing for evaluation	28

3.5.1	Benchmarking	28
3.5.2	In-memory implementation	28
3.6	Repository overview	28
3.6.1	Source code	28
3.6.2	Tests	29
3.7	Summary	29
4	Evaluation	30
4.1	Evaluation setup	30
4.2	Acceptance testing	30
4.3	Comparative evaluation	30
4.4	Scalability	31
4.4.1	Transitive Closure	32
4.4.2	Strongly Connected Component and Reverse Same Generation . . .	33
4.4.3	Points-to analysis	34
4.4.4	Experiments on AWS EC2 instances	34
4.5	Fault tolerance	35
4.5.1	Correctness	36
4.5.2	Throughput	36
4.6	Summary	38
5	Conclusions	39
5.1	Achievements	39
5.2	Future work	39
5.3	Lessons learned	40
	Bibliography	41
	Appendices	45
A	Coordinator-Worker interaction	46
A.1	Coordinator server RPC handling	46
A.2	Worker task computation	47
B	AWS and Docker Specification	49
B.1	AWS EC2 specification	49
B.2	Docker container configuration	49
C	Software libraries versions	53
C.1	Software tools used in this project	53
D	Project Proposal	

List of Figures

2.1	Precedence graph for a non-stratifiable program in Listing 2.2.	7
2.2	Precedence graph of the unreachability program.	7
2.3	Gantt chart for this project. Bars are the scheduled time, and colours indicate the actual time spent on the task.	11
3.1	Erlog engine structure.	13
3.2	Call graph of the engine evaluating TC. Profiled using <code>fprof</code> [22] and visualised with <code>erlgrind</code> [16] and <code>qcachegrind</code> [56].	16
3.3	Computations to reach fixpoint.	19
3.4	An example of the <i>simple distributed join</i> algorithm. Arrows indicate sources of the tuples that we are going to join. Crossed-out tuples are those that have been generated before.	21
3.5	When all tasks are in progress or finished, but the current stage is not completed yet, the coordinator asks workers to wait.	23
3.6	Coordinator handles failed workers.	24
3.7	Worker 1 requests a task from the coordinator, and the coordinator finds that task 1 (assigned to worker 2 currently) would have been finished sooner if it were given to worker 1 now. Hence it assigns it to worker 1. Green bars indicate the progress of tasks.	26
3.8	Directory structure and module dependencies of the project.	29
4.1	Comparing the distributed engine with other execution modes.	31
4.2	Comparing the flame graph of two implementations. Generated using <code>eflame</code> [44].	32
4.3	TC scalability on two different graphs.	33
4.4	RSG and SCC scalability.	34
4.5	Points-to analysis program throughput.	35
4.6	AWS and local Docker container throughput comparison.	36
4.7	Network throughput of containers during computation.	36
4.8	TC program throughput in the presence of worker failures. Failure rate is defined as the percentage of workers that will fail.	37
4.9	TC program throughput when there are stragglers. The engine maintains scalability with the help of a scheduler.	38

List of Listings

2.1	The all-pairs reachability program.	4
2.2	Program with no unique fixpoint.	6
2.3	The unreachability program.	7
2.4	Stratification of the unreachability program.	8
3.1	Erlog syntax, in BNF form.	14
4.1	Strongly Connected Component program.	33
4.2	Reverse Same Generation program.	33
4.3	A simple context-insensitive, field-sensitive points-to analysis [51]	34
A.1	Coordinator handlers for RPCs.	47
A.2	Worker requesting tasks from coordinator and does computation.	48
B.1	Docker compose file.	52

Chapter 1

Introduction

In this dissertation, I present a way of achieving data-parallel processing in datalog programs. I exploited the data-parallelism among the Relational Algebra (RA) operators as primitives of datalog query processing. This idea is implemented as a distributed datalog engine, *Erlog*, by taking inspiration from various distributed computation frameworks. We shall see that the parallel nature of RA primitives indeed allows us to achieve strong scalability in datalog program evaluation.

1.1 Motivation

Datalog is a declarative logic programming language [47]. It is rooted in the database systems community and was first developed in the eighties and early nineties when systems such as Coral [50] and LDL++ [7] were built. Some ideas from these early research prototypes made it into mainstream commercial database systems [30]. However, datalog research has since been inactive for a long time due to the lack of compelling applications [36]. In recent years, there has been an increase in the use of recursive query languages to do information extraction, program analysis and declarative networking [30]. Datalog has regained its popularity due to these emerging applications.

One particular example of datalog application is query processing in database systems. It is similar to SQL, but allows recursive queries that are not part of the core SQL. Queries like Transitive Closure (TC) allow problems such as bill-of-material to be solved by the database system itself [11]. Since we are working with database systems, we are faced with an increasing amount of data, and wish to scale our database systems accordingly. Although Moore’s law allows us to scale machines vertically by just waiting for advances in hardware, other bottlenecks such as power efficiency are now becoming the limiting factor [48]. Therefore horizontal scaling, such as distributed computation on a sharded dataset is becoming more popular.

The Erlog engine implements datalog queries as Relational Algebra operations. These operations are mostly *coarse-grained*, which means they typically do not involve fine-grained access to particular memory addresses. We can therefore achieve data-level parallelism by making these operations themselves data-parallel. We distribute our work to multiple workers and ask them to perform these data-parallel RA operations on the partitioned data independently, and the final result would be their aggregation. This approach turns out to scale well in the number of workers on different kinds of programs (§4.4).

Erlog models the distributed computation in a *coordinator-worker* fashion, drawing inspirations from various distributed programming models, such as MapReduce (MR) [15] and Resilient Distributed Datasets (RDD) [58]. These computation frameworks are designed for batch processing of data and, therefore, are suitable for implementing this engine. Moreover, they also provide us with fault tolerance against crash-stop workers and maintain scalability when there are slow workers in the cluster (§ 4.5).

1.2 Related areas

There are already well-built, highly-efficient datalog engines such as Soufflé [52] and LogicBlox [38]. These engines provide high performance on a single node using thread-safe data structures such as a concurrent B-tree [40]. However, they do not support distributed computation.

There are also attempts to build a data-parallel version of datalog, such as Google’s Yedalog [13], which aims to simplify large-scale knowledge exploration through the conciseness of logical programming. With Yedalog, a programmer can express his query, such as counting a particular word, in a logical paradigm. Yedalog then attempts to map these logical rules to MR jobs. These jobs are compiled into Flume pipelines [12] and executed in parallel by Yedalog’s batch backend. Apart from Yedalog, recent advances in Balanced Parallel Relational Algebra (BPRA) has also seen application in MPI based data-parallel datalog engines [29].

This project takes a similar approach to Yedalog but focuses on addressing recursive queries such as TC. It is also worth noting that Yedalog is proprietary. Therefore the exact method and effectiveness of Yedalog are unclear (as of 2022).

1.3 Dissertation structure

Chapter 2 prepares the background knowledge for the implementation and evaluation of the engine. Chapter 3 then discusses the implementation of Erlog features and various design choices. Chapter 4 evaluates the engine in terms of scalability and fault tolerance before Chapter 5 concludes the dissertation and lists some possible future work.

Chapter 2

Preparation

This chapter describes the background knowledge needed to implement the engine (§ 2.1), including the semi-naïve algorithm (§ 2.1.3) and distributed computation frameworks (§ 2.1.5). This is followed by an analysis and refinement of the originally proposed work (§ 2.2). We will then talk about various software engineering techniques to aid the development cycle (§ 2.3) before declaring the starting point (§ 2.4).

2.1 Background knowledge

2.1.1 The datalog language

A datalog program is a collection of datalog *rules*, and each *rule* has the form:

$$A \text{ :- } B_1, B_2, \dots, B_n.$$

where *A* is called the *head* of the rule, and the *B*'s constitute the *body* of the rule. *A* and *B* are called *atoms* in datalog (notice this is a different use of the term *atom* from Prolog, where atoms are just constants), and each *atom* has the form `pred_sym(term, term, ...)`, where `pred_sym` is the *predicate symbol*, or the name of the atom, and *terms* are the arguments. Each *term* can either be a *constant*, which starts with a lower case letter, or a *variable*, which starts with a capital letter. For example, `reachable(X, Y)` is an atom, with the predicate symbol `reachable` and two arguments, *X* and *Y*, both of which are variables.

2.1.2 Fixpoint-theoretic semantics of datalog

There are three different but equivalent ways of looking at the semantics of the core datalog language: *model-theoretic* semantics, *proof-theoretic* semantics and *fixpoint-theoretic* semantics [30]. We will be focusing on the last one since it serves as our implementation strategy for the datalog engine.

The fixpoint semantics is based on the *immediate consequence* operator T_P :

$$T_P : \text{inst}(\text{sch}(P)) \rightarrow \text{inst}(\text{sch}(P))$$

To explain the concept of the *immediate consequence* operator, we first need to define *edb* and *idb* relations. An *extensional relation* occurs only in the body of a rule, such as `link`, and is often viewed as given from the input. On the other hand, an *intensional*

2.1. BACKGROUND KNOWLEDGE

relation occurs in the head of a rule and hence does not appear in the source database. Rather, only the “intension” of how to compute it is given by the rules. `reachable(X,Y)` from the previous section is an example of an *idb* relation. The *extensional database* (*edb*) of a program P , denoted as $\text{edb}(P)$ is the set of all *extensional relations*, while the *intensional database* (*idb*): $\text{idb}(P)$ consists of all the *intensional relations*. Intuitively, the *edb* predicates are given as the source database, and the *idb* predicates are derived based on the rules and the source.

Now we define *immediate consequence* as follows: An atom A is an *immediate consequence* of a program P and database instance I if A is a ground *edb atom* in I or A is in a rule in P of the form $A :- B_1, \dots, B_n$, and B_1, \dots, B_n are ground instances in I . And then we can define $A \in T_P(I)$ if and only if A is an *immediate consequence* of P and I .

Under this semantics, executing a program P on an instance I involves repeatedly applying the *immediate consequence* operator T_P until we reach a *fixpoint*, i.e. $T_P(I) = I$. There may be multiple fixed points for T_P , and we seek the least one: $\text{fix}(T_P)$. The least fixed point always exists for any core datalog program and source instances due to Knaster-Tarski’s theorem [53]. Moreover, it is also equal to $P(I)$, the minimum model ($I' \supseteq I$ and satisfies all constraints in P) of the program. This follows from the fact that $P(I)$ is a model and T_P is monotonic. The details of this proof can be found in Abiteboul et al. [1]. For now, we just need to appreciate the fact that applying T_P repeatedly to a core datalog program always terminates with the results we need.

This semantics is useful because it gives us a way of implementing the evaluator constructively. We can convert the program into its corresponding *immediate consequence* operator and then apply this operator to the input instance until we reach a fixpoint. Indeed, this is how most datalog engines are implemented and is the implementation strategy for this project.

2.1.3 Recursive query processing

Now that we have decided to use the *fixpoint* evaluation strategy, we can look at two ways of implementing it: naïve and semi-naïve evaluation.

The naïve method

The naïve method repeatedly applies rules to the database instance to derive new tuples. We keep doing this until no new tuples can be derived. Table 2.1 shows an example of computing the fixpoint of the program given in Listing 2.1.

```
reachable(X, Y) :- link(X, Y).
reachable(X, Y) :- reachable(X, Z), link(Z, Y).
```

Listing 2.1: The all-pairs reachability program.

X	Y
a	b
b	c
c	d

(a) Initial input **link** predicates.

X	Y
a	b
b	c
c	d

(b) **reachable** predicates after iteration 1.

X	Y
a	b
b	c
c	d
a	c
b	d

(c) **reachable** predicates after iteration 2.

X	Y
a	b
b	c
c	d
a	c
b	d
a	d

(d) **reachable** predicates after iteration 3.

Table 2.1: An example of computing the fixpoint of a query, where we reach the fixpoint in iteration 4. Coloured rows are new tuples derived.

The semi-naïve method

While the naïve evaluation method is simple to understand and implement, it suffers from redundant computations. Remember that the immediate consequence operator is applied to all the tuples that have been derived at each iteration. This results in tuples generated in previous iterations being computed again in the current iteration. The semi-naïve evaluation strategy builds on top of the naïve strategy by keeping track of a set of relations $\Delta_{\text{predicate}}^i$. Intuitively, these are the tuples newly generated in the i th iteration. Based on the observation that new tuples can only ever be generated if the tuples used to generate them are new, in the $(i + 1)$ th iteration, only new tuples in the Δ^i set will be used to compute the immediate consequence.

For example, looking at the program in Listing 2.1, the semi-naïve “version” of the program would look like the program below, where each $\Delta_{\text{predicate}}^i$ represents the set of tuples for `predicate` generated in the i th iteration.

$$\begin{aligned}
 \Delta_{\text{reachable}}^1(X, Y) &:- \text{link}(X, Y) \\
 \Delta_{\text{reachable}}^{i+1}(X, Y) &:- \Delta_{\text{reachable}}^i(X, Z), \text{link}(X, Y)
 \end{aligned}$$

2.1.4 Negation

The monotonicity of the core datalog comes with the restriction that we cannot express non-monotone queries such as negation. The language semantics does not extend naturally to negated datalog programs. Consider, for example, the program in Listing 2.2, which has two least fixed points, i.e. $\{p\}$ and $\{q\}$. However, in practice, we might still want the convenience of negations in our queries. For example, we might want to find all the nodes in a graph that cannot reach each other. As part of the extension of this project, we attempt to incorporate negation into *rule bodies* by first looking at a restricted case of having only negated *edb* atoms and then extending this to a more general case with stratification.

$$\begin{aligned} p &:- !q. \\ q &:- !p. \end{aligned}$$
Listing 2.2: Program with no unique fixpoint.

Semipositive datalog

We first define the concept of an *active domain*. The *active domain* of a database instance I is the set of all constants in I .

We define a datalog[−] program to be *semipositive* whenever a negative predicate $\neg r'(X)$ appears in the body of a rule, $r'(X) \in \text{edb}(P)$. This means that we are only allowed to have negative *edb* predicates. With this constraint, we consider a rule

$$\text{diff}(X) :- r(X), \neg r'(X).$$

in which r' is an *edb* predicate, we can simply define $\bar{r}'(X)$ as a new *edb* predicate that holds the complement of $r'(X)$ with respect to the *active domain*. Now we can replace $\neg r'$ with \bar{r}' and apply the familiar core datalog semantics.

Straified negation

Semipositive datalog restricts our use of negation on *edb* predicates only. If we consider a datalog program where some predicates are defined by a subset of its rules, then other programs can treat them as *edb* predicates once they are computed based on these subsets of rules. This is the idea behind *stratified negation*.

We define a *stratification* of a datalog[−] program P to be a sequence of datalog[−] programs P_1, \dots, P_n such that for some mapping σ from $\text{idb}(P)$ to $[1 \dots n]$,

1. P_1, \dots, P_n is a partition of P .
2. For each *idb* predicate r , all the rules in P defining r are in $P^{\sigma(r)}$.
3. If $r(u) \leftarrow \dots r'(v) \dots$ is a rule in P , and r' is an *idb* relation, then $\sigma(r') \leq \sigma(r)$.
4. If $r(u) \leftarrow \dots \neg r'(v) \dots$ is a rule in P , and r' is an *idb* relation, then $\sigma(r') < \sigma(r)$.

Intuitively, stratification gives us an order to evaluate a datalog program: we first evaluate those without negation, and then look at rules with negation, with the previously obtained results as our *edb* predicates.

We note that not all programs are *stratifiable*, for example, the program in [Listing 2.2](#). We cannot find an ordering to execute the program that can satisfy the constraint above. To determine whether a datalog[−] program is *stratifiable*, we compute its *precedence graph* by taking the following steps:

1. Construct nodes for each *idb* relation in the program.
2. If $r(u) \leftarrow \dots r'(v) \dots$ is a rule in P , then add an edge from r' to r labelled with $+$.

CHAPTER 2. PREPARATION

3. If $r(u) \leftarrow \dots \neg r'(v) \dots$ is a rule in P , then add an edge from r' to r labelled with $-$.

Then the program P is stratifiable if and only if there is no cycle in the *precedence graph* containing a negative edge [1]. Intuitively, this means that we disallow recursive applications of negation, which is also one of the limitations of stratified negation.

The example program in Listing 2.2 has the *precedence graph* G in Figure 2.1. We can see that it contains a cycle with negative edges in it and hence is not stratifiable.

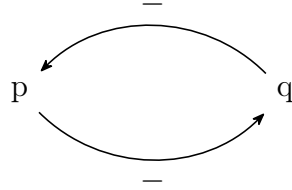


Figure 2.1: Precedence graph for a non-stratifiable program in Listing 2.2.

The *precedence graph* G can also help us compute the *stratification* of a program. The following algorithm demonstrates this.

1. Compute the Strongly Connected Component (SCC) of the *precedence graph*, and let these components be our strata.
2. Perform a topological sort on the strata obtained, respecting the total ordering of the original *precedence graph*. This gives us one stratification of the program.

Note that a program might have multiple *stratifications*. It turns out all of them are equivalent [1], i.e. they all lead to the same results when applied to an input.

We illustrate the algorithm further by considering an example stratifiable program in Listing 2.3, and its *precedence graph* in Figure 2.2.

```
reachable(X,Y) :- link(X,Y).
reachable(X,Y) :- link(X,Z), reachable(Z,Y).
node(X) :- link(X,Y).
node(Y) :- link(X,Y).
unreachable(X,Y) :- node(X), node(Y), !reachable(X,Y).
```

Listing 2.3: The unreachability program.

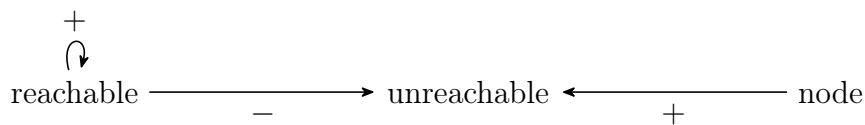


Figure 2.2: Precedence graph of the unreachability program.

We compute the SCC and topologically sort the graph to obtain a stratification in Listing 2.4.

```

P1: reachable(X,Y) :- link(X,Y).
    reachable(X,Y) :- link(X,Z), reachable(Z,Y).

P2: node(X) :- link(X,Y).
    node(Y) :- link(X,Y).

P3: unreachable(X,Y) :- node(X), node(Y), !reachable(X,Y).

```

Listing 2.4: Stratification of the unreachability program.

Stratified negation provides us with a way of evaluating a negated datalog program. Moreover, it has the same property as the core datalog in that $P^{\text{strat}(I)}$ is a minimal model containing I [1].

2.1.5 Distributed framework

In § 1.1, we mentioned that Erlog borrows ideas from various distributed computation models and frameworks. In this section, we compare these frameworks in more detail, followed by a discussion of how they relate to this project.

MapReduce (MR)

MapReduce is a programming model and an associated implementation for processing and generating large datasets [15]. As its name suggests, the user supplies a *map* function that would process a key/value pair and generate intermediate key/value pairs, which are then reduced by the *reduce* function.

A MR job has three phases [12]:

1. The *Map* phase reads in the input of a collection of values or key/value pairs and apply the user-defined mapping function *Mapper* to the input data.
2. The *Shuffle* phase takes the results produced at the *Map* phase and groups them according to their keys. It then outputs each distinct key with a stream of values to the next phase.
3. The *Reduce* phase takes the key and a sequence of values associated with the key and applies the reduce function *Reducer*. The *Reducer* typically aggregates the values associated with each key and emits the aggregated output to the output sink.

This model is quite relevant to this project because in semi-naïve evaluation, we have a series of iterations, and we can treat computations in each iteration as one MR job. The input relations are partitioned, and every worker then carries out one iteration of computation on the given input. Results produced from the current stage are re-partitioned to prepare for the next iteration, which corresponds to the *shuffle* phase of MR. Therefore the computation of a datalog program can then be viewed as a sequence of MR jobs.

Resilient Distributed Datasets (RDD) and Spark

The Resilient Distributed Datasets is a memory abstraction that allows programmers to perform in-memory computations in a fault-tolerant manner [58]. Spark is an open source implementation of RDD [5]. An RDD is a big dataset sharded across memories of multiple machines. Programmers can perform distributed computation by specifying a series of *transformations* on the RDD, which constitute the *lineage graph*: a computation graph specifying how to transform one RDD to another.

RDD is particularly good at iterative algorithms, for example, the PageRank algorithm. This is because these algorithms contain multiple iterations, which typically require multiple MR jobs to compute and incur a high IO cost. On the other hand, the RDD abstraction allows us to keep intermediate results in memory.

Another advantage of the RDD abstraction is its fault tolerance. A single MR job is inherently fault-tolerant. RDD offers its fault tolerance through the availability of its initial input data as well as the required computations (the *lineage graph*). Furthermore, when a *transformation* involves communication across workers, RDD would persist results to prevent one worker failure from causing restart of the whole chain.

Framework for this project

Looking at these two frameworks, MR is simple and expresses the need for one iteration of the semi-naïve evaluation. RDD utilises an in-memory data structure and is efficient when the workflow involves iterative computations, matching the need to repeatedly apply the T_P operator while evaluating a datalog program.

However, RDD itself is a quite general computation framework that can handle lots of different workflows. For this project, we are only interested in building an engine that can do distributed computation for datalog programs, so there is no need to reproduce an RDD framework. Moreover, in order to achieve fault tolerance, RDD materialises intermediate results of *wide dependencies*: computations that involve communication across workers. In our case, the result of *every* iteration is a *wide dependency*, which means RDD would need to materialise results produced by workers at the end of every iteration. This makes it more similar to a sequence of MR jobs.

In conclusion, this project implements an MR pipeline to evaluate a datalog program with multiple workers, persisting intermediate results produced by each worker at each stage to ensure fault tolerance.

2.1.6 Erlang features

Erlang is a functional language commonly used to build massively scalable soft real-time systems with high availability requirements [26]. A number of features that are desirable for this project are discussed below.

Erlang’s concurrency model allows programmers to spawn lightweight processes called *actors* and use *message passing* for inter-process communication. This is useful when

writing concurrent applications.

Erlang’s *fault tolerance* is based on the observation that failures will happen no matter what, and it is rarely practical to get rid of all errors in a program or a system [35]. Therefore the recommended way of programming is to let failing processes *crash* and other processes detect the crash and fix them [6]. This is where Erlang’s *supervision tree* [54] comes into place. A supervisor “supervises” its child process(es) and when it detects failures, it will attempt to restart it. This provides useful functionality for this project to achieve fault tolerance.

Erlang/OTP also provides abstractions for commonly used *behaviours*. In particular, the `gen_server` behaviour models client-server principles [25] and allows the programmer to implement a simple set of callback functions to get complete server behaviour (similar to interfaces in Java). This is useful for modelling the behaviour of coordinators and workers in our engine.

Last but not least, Erlang/OTP has good support for *distributed programming*, including creating Erlang VM/nodes, establishing connections between nodes and remotely spawning processes. Its first-class support for remote procedure calls [19] is particularly useful for writing a distributed application, such as Erlog.

2.2 Requirements analysis

According to the original project proposal (Appendix D), the baseline requirement for this project is to have a distributed engine that can evaluate a transitive closure program. Therefore we need to implement an engine that does distributed computation. Based on that, we wish to examine two aspects of this engine:

Scalability We want to evaluate the change in throughput as we increase the number of workers. This is done by adding more workers to the engine and plotting a graph of throughput against the number of workers.

Fault tolerance We wish to look at the engine’s response to abnormal worker behaviours. This can be done by randomly killing workers and checking the correctness of the output against the serial engine. The impact of worker failures and slow workers on scalability can also be studied.

Apart from computing the core transitive closure program, to make the engine more user-friendly, it should also have a compiler frontend: a lexer and a parser. We might also consider support for programs other than transitive closure, such as strongly connected components, points-to analysis and negated datalog programs (extension).

In terms of extensions, I scheduled three out of the four originally proposed extensions as the last one (implementing BPRA) is likely to take a long time and only considered when I am ahead of schedule. This is shown in the Gantt chart (Figure 2.3).

2.3 Software engineering techniques

2.3.1 Development model

This project follows the spiral development model [10], which offers a risk-driven way of developing the application. The riskiest part of the project: distributed evaluation is identified first, along with its dependent features. They are implemented as soon as possible by compromising some level of the robustness of the engine, and possibly skipping some tests. Initially I tried to focus only on the transitive closure program to quickly get a working distributed engine. I also used the Kanban board [2] as a project management tool to keep track of deadlines and different features that I wish to implement at various stages.

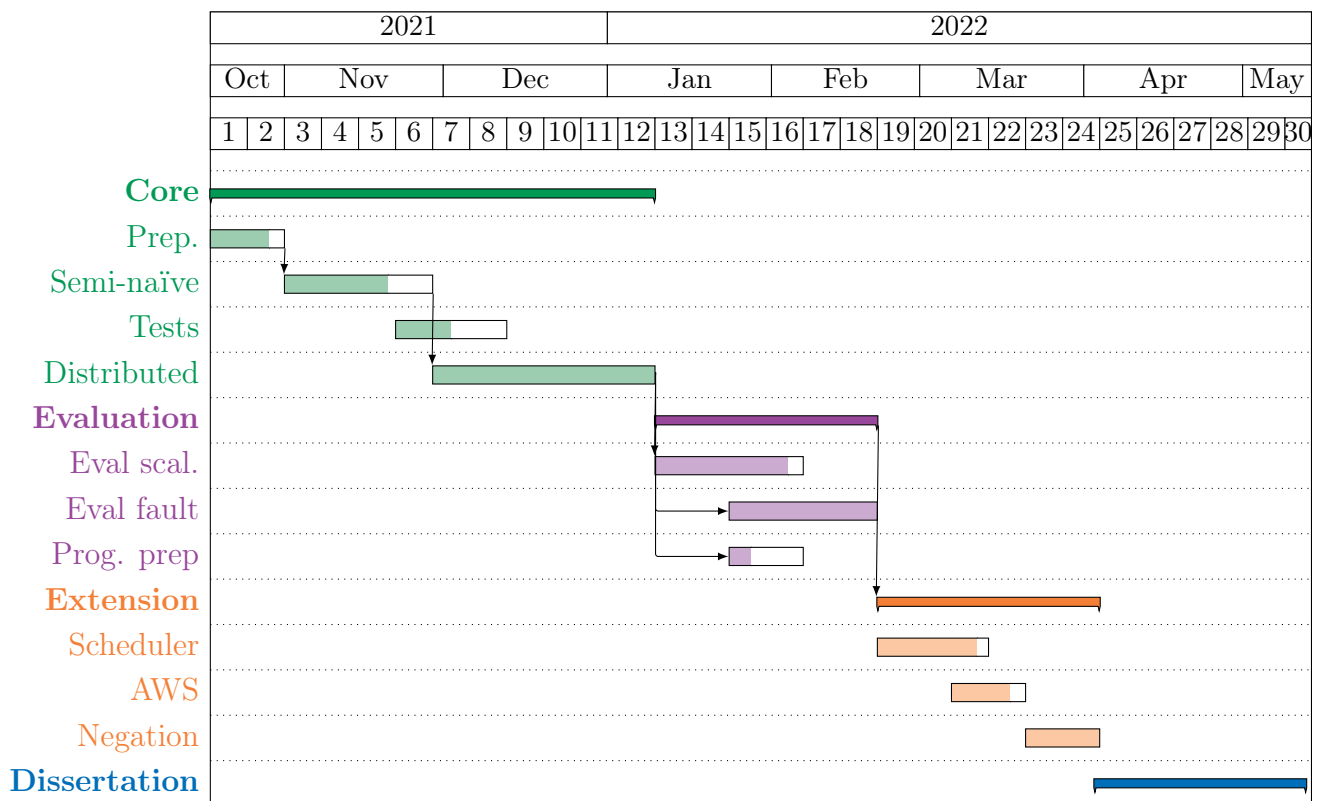


Figure 2.3: Gantt chart for this project. Bars are the scheduled time, and colours indicate the actual time spent on the task.

2.3.2 Backup, version control and testing

This project is version controlled using `git`, backed up to Github after each commit, synchronised to Google Drive and an external hard disk fortnightly.

The datalog engine is tested with Erlang’s unit testing framework, EUnit [21], and Common Test [17] for larger-scale tests, including both white-box and black-box testing.

2.3.3 Software license

The source code for this project is publically available under the Apache License 2.0 license [4]. I chose this license because it is a permissive license that grants both private and patent use. It also allows the work and its modifications to be distributed under different terms.

2.4 Starting point

This project is implemented from scratch. I had no prior experience working with Erlang apart from reading the first few chapters of *Learn You Some Erlang for Great Good!* [55] and did some exercises in the book during the summer holiday.

I gained familiarity with compiler design and logic programming language through studying the Part IB Compiler Construction and Prolog course.

My knowledge of distributed systems primarily comes from the Part IB Concurrent and Distributed Systems course. The Part II Unit of Assessment Cloud Computing also gives more perspective, but this happened during Lent term, which was after I had started my project. I also had experience writing a toy MapReduce application as a word count application for personal interest. Similar concepts can be borrowed, but no existing codebase was used.

2.5 Summary

In this chapter, we looked at various background knowledge needed to implement the engine, including the fixpoint semantics of the datalog language (§2.1.2) and its associated evaluation methodology (§2.1.3), different distributed computation frameworks (§2.1.5), and the language features provided by Erlang (§2.1.6). In the next chapter, we shall implement Erlog with this knowledge.

Chapter 3

Implementation

This chapter discusses the implementation of the distributed engine. We start by giving a high-level structure of this chapter diagrammatically (§3.1) before defining the syntax of the Erlog language (§3.2). We then look at the algorithm for evaluating a datalog program on a single node (§3.3), as well as on a distributed cluster of nodes (§3.4). Finally, an overview of the repository is given (§3.6).

3.1 Overview

Figure 3.1 shows a diagram of the components of the engine. A programmer writes some datalog code (§3.2) and passes it on to the coordinator of the distributed engine. The coordinator (§3.4.4) distributes its work (§3.4.2) to all the workers (arrows from the coordinator to worker 2 onwards are omitted for simplicity, in reality the coordinator controls all workers) and oversees their execution. Workers (§3.4.5) then carry out computations at each stage with the semi-naïve algorithm (§3.3).

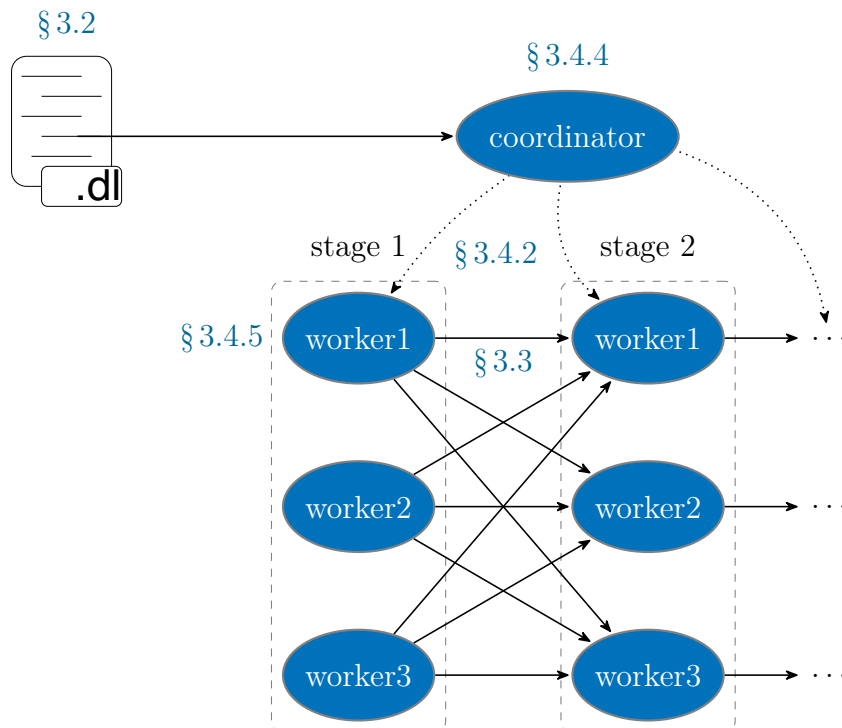


Figure 3.1: Erlog engine structure.

3.2 Erlog syntax

We now define the syntax for Erlog. There is nothing special about this language requiring novel syntax, so we will follow the conventional datalog syntax. This is shown in [Listing 3.1](#).

```

prog -> rule rules
      |  $\epsilon$ 
rule -> rule_head :- rule_body.
rule_head -> atom
rule_body -> atoms
atoms -> atom atoms
        | atom
atom -> const(terms)
terms -> term, terms
term -> const
       | var
term -> var

```

Listing 3.1: Erlog syntax, in BNF form.

To implement a parser for Erlog, the syntax above is translated into `yacc` and `leex` code, which are Erlang’s parser and lexer generator [23, 24] that largely resemble the famous `yacc` and `lex` [45].

3.3 Single node evaluation

This section looks at the implementation of the engine on a single node with RA primitives. We will look at the semi-naïve algorithm for finding the fixpoint of the *immediate consequence* operator T_P , followed by the *stratified negation* technique, which allows us to evaluate negated datalog programs.

3.3.1 Relational Algebra (RA)

Many high-performance datalog solvers employ specialised RA operations to implement datalog rules [29]. Indeed, any single datalog rule can be implemented with RA operations [28].

We start by defining the Relational Algebra as follows [32]:

$$\begin{aligned}
 Q &::= R \text{ base relation} \\
 &| \sigma_p(Q) \text{ selection} \\
 &| \pi_{\mathbf{X}}(Q) \text{ projection} \\
 &| \rho_M(Q) \text{ renaming} \\
 &| Q \times Q \text{ product}
 \end{aligned}$$

CHAPTER 3. IMPLEMENTATION

where p in σ_p is the predicate for selection, \mathbf{X} in $\pi_{\mathbf{X}}$ is the set of columns we are projecting, M in ρ_M consists of a renaming map such as $\{A_1 \mapsto B_1 \dots\}$.

3.3.2 Database operation primitives

We now look at how to implement datalog rules in Relational Algebra. We will use a few examples to illustrate this and will be focusing on intuition rather than formality.

Projection

We consider a datalog rule:

```
reachable(X, Y) :- reachable3(X, Z, Y).
```

We can see that this rule can be implemented as a projection operator and we can construct a new relation from the projected tuples, i.e.

$$\text{reachable}(X, Y) = \pi_{X, Z}(\text{reachable3}(X, Z, Y))$$

Product

Now consider another datalog rule:

```
R(A, B, C, D) :- P(A, B), Q(C, D).
```

We can see that this can be implemented in RA primitive as a product of two relations:

$$R = P \times Q$$

Natural join

Join is probably the most important primitive used in the engine. First, let us define the natural join operator \bowtie : if we are given two relations $R(\mathbf{A}, \mathbf{B})$ and $S(\mathbf{B}, \mathbf{C})$, we can then define the natural join of R and S as

$$R \bowtie S = \{t \mid \exists u \in R, v \in S, u.[\mathbf{B}] = v.[\mathbf{B}] \wedge t = u.[\mathbf{A}] \cup u.[\mathbf{B}] \cup v.[\mathbf{C}]\}$$

Again, we consider an example rule:

```
reachable(X, Y) :- link(X, Z), reachable(Z, Y).
```

We see that the rule is of a similar form to the natural join we have just defined, and if we were to write it in RA operators, it would look like:

$$\text{reachable} = \pi_{X, Y}(\text{link} \bowtie_{2=1} \text{reachable})$$

where 2 and 1 are one-based indices of the arguments in the tuple.

3.3. SINGLE NODE EVALUATION

Finally, to simplify implementation, we can see that the product operation mentioned above is just a particular case of the join operation in that the predicate for join is true, and the following projection is the identity operation.

One decision to make is to choose between *join-before-select* and *select-before-join*. The former is to gather everything we know about the two relations and then make sure the arguments we are joining match with each other. The latter tries to find valid atoms for the first relation, substituting these variables to the arguments we will join in the second relation, and then tries to find atoms of the second relation.

It usually does not matter which one we choose. *Join-before-select* was implemented first and profiled. Figure 3.2 shows a segment of the call graph and the proportion of time each function takes. We see that the `dbs:join_one/5` function (on the left branch) accounts for 17.14% of the time. This implementation takes $\mathcal{O}(n^2)$ time as we need to examine every possible pair of predicates and see if they can be joined. To speed it up, we can borrow some ideas from the *select-before-join* approach: scan the first database to be joined and store the arguments of the atoms on which we will join in a set data structure. This allows fast lookup when we scan the second database to be joined and therefore reduces the time complexity to $\mathcal{O}(n)$ by using additional space.

Since join is used frequently in the engine, it is worth trading space for time. Indeed, a benchmarking shows a 25% reduction in execution time with the improved join algorithm on a graph of 200 nodes.

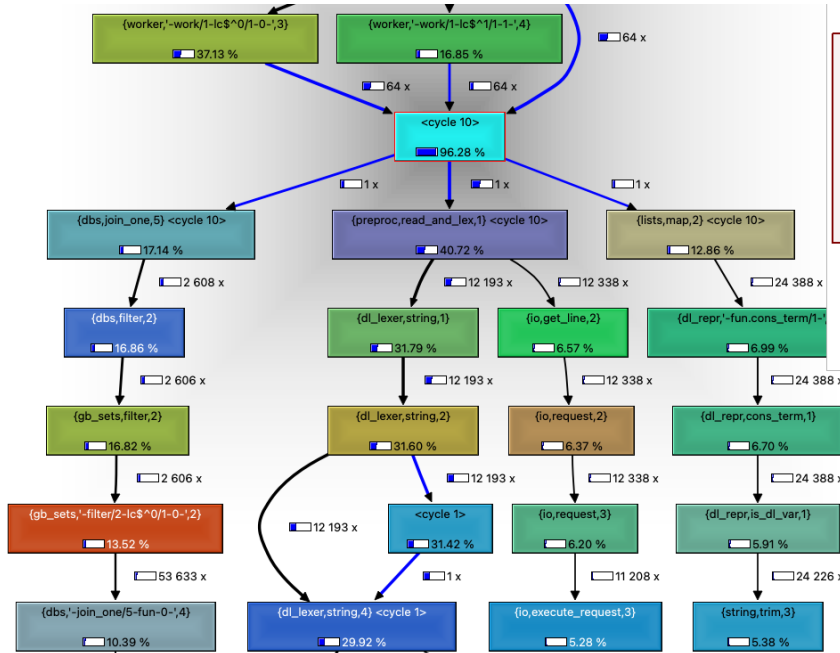


Figure 3.2: Call graph of the engine evaluating TC. Profiled using `fprof` [22] and visualised with `erlgrind` [16] and `qcachegrind` [56].

CHAPTER 3. IMPLEMENTATION

3.3.3 Implement datalog as Database operations

Now we are ready to put all of these operations together to implement an `eval_one_rule/3` function which will evaluate one rule given the database instance. We can define a function that takes one rule, pattern matches it with all the cases mentioned above and chooses the appropriate RA operation to apply.

```
-spec eval_one_rule(dl_rule(), dl_program(), dl_db_instance()) -> dl_db_instance().
```

Then we can define another function called `imm_conseq/2` that will go through each rule in the program and apply them one by one.

```
-spec imm_conseq(dl_program(), dl_db_instance()) -> dl_db_instance().
```

One design question to ask is how to represent a database instance in Erlang. We know that a database instance is a *set* of atoms, so conceptually we would want to use a set-like data structure. However, Erlang has many such data structures, including `ordsets`, `sets`, `gb_sets` and `sofs` [55]. `gb_sets` is implemented using General Balanced Trees [3], making most operations on it logarithmic. `sets` is implemented using dynamic hashing techniques [33], giving constant asymptotic insertion time. A general rule of thumb to follow is to use `gb_sets` [34]. In this project, some benchmarking comparing the performance difference between `sets` and `gb_sets` shows that `gb_sets` is slightly faster. Therefore it is used for this purpose.

3.3.4 Semi-naïve evaluation strategy

In §2.1.3, we talked about the methodology of semi-naïve evaluation. This section goes into more detail on how to implement it. We start by looking at the pseudocode in Algorithm 1. Before applying our immediate consequence operator repeatedly to the delta database, we first need to do one iteration. This is to “bootstrap” the semi-naïve evaluation process by computing the initial delta database instance. Entering the loop, we first put the Δ_S^{i-1} set from the last iteration ($i - 1$) to our accumulated results S^i , then apply the rules associated with each *idb* predicate S , i.e. P_S , on the delta *idb* predicates (Δ_T) making up the body of the rule defining S . This is repeated until no new tuples are generated, which means we have reached the fixpoint.

Another practical subtlety here is that we need to refine our notion of an *edb* predicate. Again, using our TC example in Listing 2.1, if we had a singleton input atom `reachable(a, b)`, then this `reachable` tuple will never be added to our iteration since our P' will not compute it. Thus, to work around this issue, we treat everything in our input as *edb* relations and put them into our first delta database instance.

3.3.5 Negation

In §2.1.4 we looked at the theory behind `datalog-`. We can implement it with the following functions (Algorithm 2). The `compute_stratification` function takes a datalog program

Algorithm 1: Semi-naïve algorithm, adapted by the dissertation author from *Foundations of Databases* [1].

input : datalog program P and input instance I
output : $P(I)$

P' are rules that do not have idb predicate in its body
 $S^0 \leftarrow \emptyset$ for each idb predicate
 $\Delta_S^1 \leftarrow P'(I)(S)$
 /* applying rules without idb predicate, i.e. one-off computation */

$i \leftarrow 1$
repeat
 for idb predicate S that has at least one idb predicate as its body and let the body idb predicates be T_1, \dots, T_l **do**
 $S^i \leftarrow S^{i-1} \cup \Delta_S^i$
 $\Delta_S^{i+1} \leftarrow P_S^i(I; T_1^{i-1}; T_l^i; \Delta_T^i) - S_i$
 /* this is computing the Δ^{i+1} based on previous deltas. T_{i-1} 's are for optimisation purposes */
 $i \leftarrow i + 1$
until $\Delta_S^i = \emptyset$ for each idb predicate S

and returns a list of programs, i.e., the *stratification*. To evaluate a negated datalog program, we execute each stratum using the semi-naïve algorithm as before and treat the results as the *edb* to the next stratum.

Algorithm 2: Stratified negation function.

function eval_stratified(program):
 for $s \leftarrow \text{compute_stratification}(\text{program})$ **do**
 $\text{idb} \leftarrow \text{eval_seminaive}(s, \text{edb})$
 $\text{edb} \leftarrow \text{idb}$
 end
 return edb

3.4 Distributed evaluation

This section extends our implementation of the single node datalog engine to make it distributed across multiple workers. We first look at an overview of the workflow (§ 3.4.1), and then discuss an important algorithm that allows us to perform distributed join (§ 3.4.2). Then we talk about how we can extend our stratified negation to be distributed (§ 3.4.3). Finally, we look at the jobs of the coordinator (§ 3.4.4) and workers (§ 3.4.5) respectively.

3.4.1 Execution overview

In Figure 3.1, we saw the high-level structure of the Erlog engine: the coordinator receives a program from the programmer and splits it into multiple tasks, which are then computed by workers. We can view our distributed algorithm as follows:

1. Treat each iteration of the semi-naïve algorithm as one stage of evaluation (§ 3.3.4).
2. Divide each stage of computation into multiple tasks (§ 3.4.2).
3. Ask workers to complete each stage, i.e. compute Δ_S^i (§ 3.4.5).
4. Repeat step 1–3 until fixpoint.

Figure 3.3 views the distributed computation as a whole. Note that each P_S^i operator (downarrow) on the Δ relation is actually carried out by multiple workers.

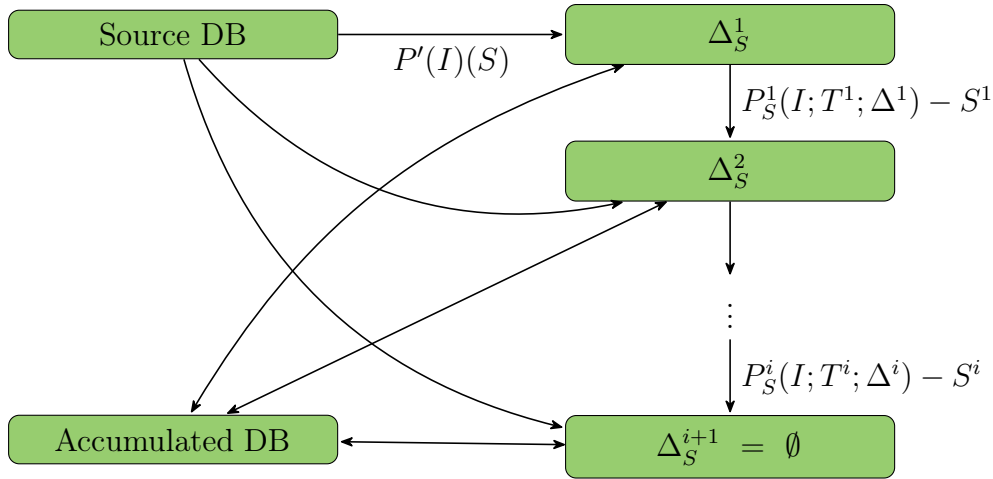


Figure 3.3: Computations to reach fixpoint.

3.4.2 Distributed RA operations and simple distributed join

We have seen how we can implement datalog rules on a single node via RA operations (§ 3.3.2). Most of these operations extend naturally to distributed computation. For example, the projection operator can be applied to arbitrarily partitioned datasets (with results aggregated at the end). However, it is not trivial to distribute the join operator. If we partition our dataset randomly, tuples that could have been joined might end up in different partitions and therefore never get joined.

This is where we need a careful way of dividing our dataset so that each partition contains the correct combination of tuples. Broadly speaking, the most commonly used method to deal with this issue is hash-based joins and hash-based distribution of relations [29]. The key idea of parallel execution strategies is that when we want to compute $R \bowtie S$, we can partition two relations R and S so that

$$R \bowtie S = R_1 \bowtie S_1 \cup \dots \cup R_n \bowtie S_n$$

To illustrate this algorithm, we can look at an example computation in [Figure 3.4](#). The input relation is partitioned into three domains. In this example, the first row is the domain $\{a, d\}$, the second is $\{b\}$ and the third is $\{c\}$. Once we have these partitions, each join operation can be done independently. Looking at the execution flow in the blue dashed box, i.e. the second domain, we see that at start, the tuple (b, c) is in the initial delta set D_1^2 and joined with the input (a, b) from R_2 to produce (a, c) . But (a, c) is not in our domain, so it is not used as the input to the next join. Instead, tuple (b, d) from D_3^2 is in the current domain, so it is joined with R_2 again to produce tuple (a, d) . Again, (a, d) is not in the current domain, but at this point, there are no new tuples from other domains that are in domain $\{b\}$, so we stop the computation. A similar process happens in other domains.

This algorithm is called *simple distributed join* algorithm. Although more advanced algorithms exist [9, 29], this one serves as a good starting point for this project.

It is not too difficult to implement such an algorithm. We need to write a function that analyses each rule to see if it needs to involve the join operator and if so, we find the arguments on which we will join. Once we have obtained these arguments, we can hash partition each atom on these arguments in the database instance. This is shown in [Algorithm 3](#).

Algorithm 3: Hash partition algorithm.

input : datalog Program P and database instance I

output : hash partitions of I according to P

part_args \leftarrow get_part_args(program)

for atom \leftarrow database_instance do

 | hash_frag(atom, part_args)

3.4.3 Stratification and parallel execution

We can extend stratified negation on a single node to distributed evaluation by computing the strata one by one. One additional parallelism we can extract here is to execute strata that do not depend on each other in parallel. This is based on the fact that there may be multiple ways to topologically sort a graph. Indeed, a program can have multiple stratifications, and all of them are equivalent (§ 2.1.4). This means two strata might not have a dependency between them and can be executed concurrently. Remember in [Figure 2.2](#), we saw that `reachable` and `node` strata do not depend on each other.

We can create a function computing the strata that can be executed together. We do this by looking for nodes in the *precedence graph* that do not have any incoming edges, putting them into a list and removing them from the graph. This is repeated until no nodes are left in the graph.

[Algorithm 4](#) shows an algorithm that takes an input program, computes its precedence graph, and returns a list, in which each element consists of a list of programs/strata that

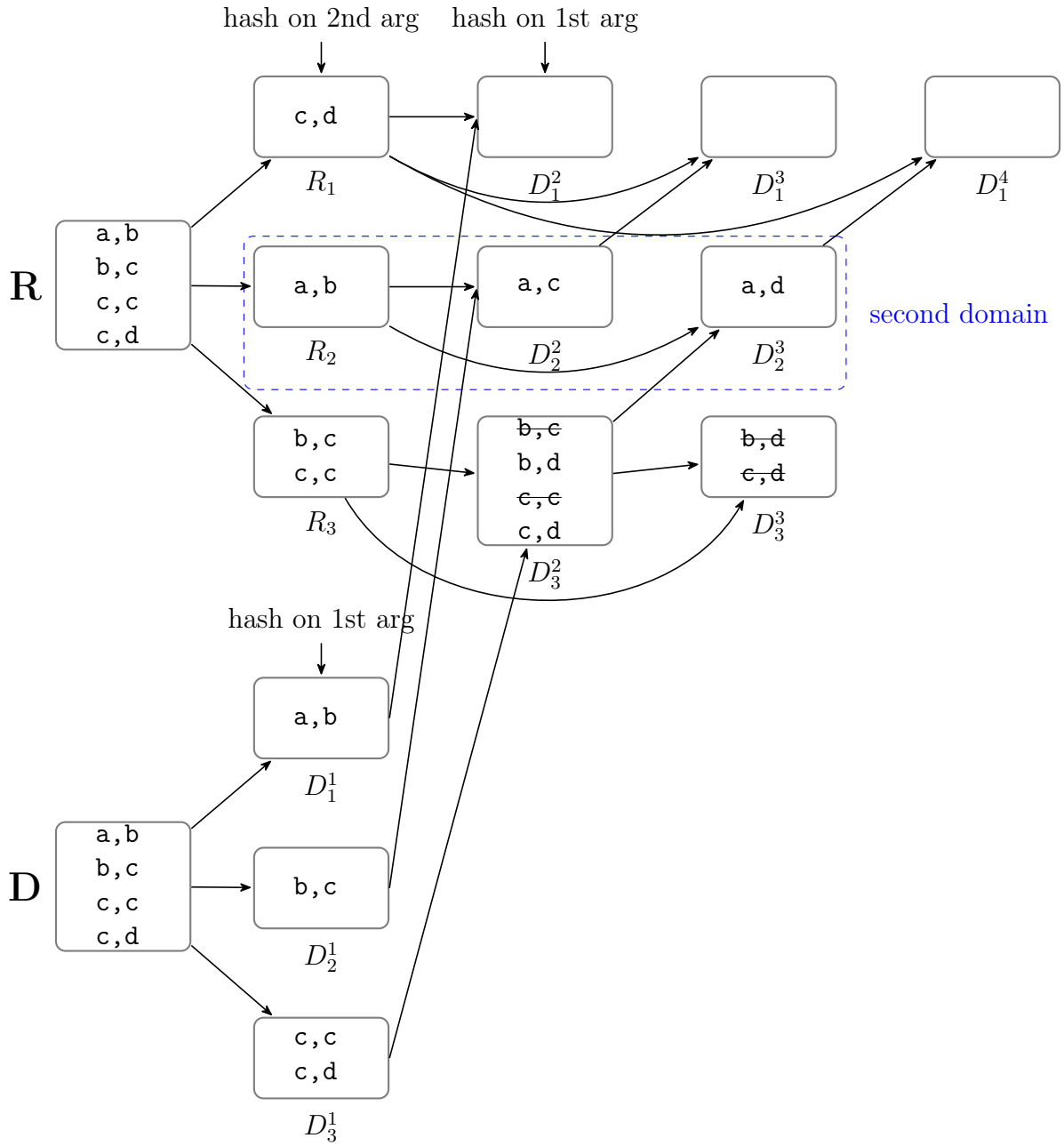


Figure 3.4: An example of the *simple distributed join* algorithm. Arrows indicate sources of the tuples that we are going to join. Crossed-out tuples are those that have been generated before.

can be executed together. It does so by repeatedly removing vertices from the graph with no incoming edges, i.e. no dependencies, and adding them to our program list. At the end of the loop, the `computed_strata` variable contains the list of programs that we are after.

3.4.4 Coordinator jobs

The coordinator serves several roles in this engine. Firstly, it needs to preprocess the input program and partition it before assigning each part to workers. It is also responsible for coordinating workers, for example, by asking some workers to wait for others when

Algorithm 4: Program computing strata that can be executed in parallel.

input : A program P

output : A list in which each element is a list of strata/programs that can be executed simultaneously

$\text{pd_graph} \leftarrow \text{compute_pred_graph}(\text{program})$

$\text{computed_strata} \leftarrow []$

while pd_graph *is not empty* **do**

$\text{verts} \leftarrow$ vertices that have no incoming edges

 prepend verts to computed_strata

 remove verts from pd_graph

there are dependencies between them. Last but not least, the coordinator needs to handle abnormal worker behaviours, and we shall consider two particular patterns: crash-stop and slow workers.

Coordinator data structure

The coordinator needs to have an internal data structure to accomplish the jobs mentioned above. Some important fields in this data structure are:

- **tasks** contains a list of tasks for **task assignment**.
- **programs** is the *stratification* of the input program for **negated** datalog programs.
- **tmp_path** is the path to temporary files storing **persistent intermediate results** of the evaluation.
- **nodes_rate** keeps track of the rate at which workers compute tasks. This is useful for **scheduling** purposes.
- **nodes_tasks** is a (hash)map that stores the assignee of each worker. This is useful when dealing with **unreliable networks** in which messages can be delayed or lost.

We shall see more details on how to use these fields in the following sections.

Task assignment and completion

Firstly, the coordinator needs to assign tasks to workers when requested. When the coordinator receives a request from a worker, it searches its list of tasks (the **tasks** field) and looks for idle ones. The task and stage number as well as the **tmp_path** field will be passed to the requesting worker. The worker can then find the input data for the task.

When the worker has finished its task, it will remotely call the **finish_task/2** method on the coordinator to ask it to update its list of tasks.

Worker coordination

In some cases, all tasks are in progress, and the current stage of evaluation has not finished yet. The coordinator then needs to ask the worker to wait for other workers. Remember that each stage of evaluation corresponds to one iteration of the semi-naïve algorithm. We cannot go to the next iteration if the current iteration is unfinished. In this case, a special wait task is assigned, which will cause the worker to sleep for some period before trying to request tasks again. This is depicted in Figure 3.5.

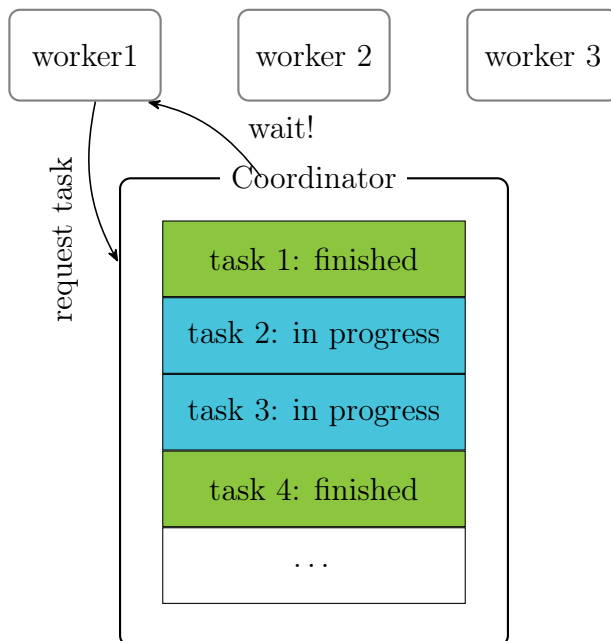


Figure 3.5: When all tasks are in progress or finished, but the current stage is not completed yet, the coordinator asks workers to wait.

Fault tolerance

It is also the job of the coordinator to offer fault tolerance. In a distributed system, there are different failure patterns we might expect from a node, such as crash-stop, crash-recovery and Byzantine workers [43].

In this implementation, we will not consider the Byzantine behaviour of workers. We assume that our worker and coordinator nodes are in the same security domain, which is reasonable since the cluster running Erlog is typically controlled by one user/organisation.

Theoretically, the fault tolerance of the engine is guaranteed by the framework used, as discussed in §2.1.5. Briefly, because we know what computations we are going to perform (the RA operations), and as we have the input data available (input database or persisted intermediate databases), we can ask another worker to recompute the results when we encounter worker failures.

The first step in making the engine tolerate worker failures is to ask the coordinator to keep track of the assigned worker for each task. Once the coordinator has detected a worker failure by sending heartbeats, it can then reschedule the task to another worker.

3.4. DISTRIBUTED EVALUATION

Because each worker works independently on one hash-domain of the input, the engine can continue running without problems. This is shown in Figure 3.6.

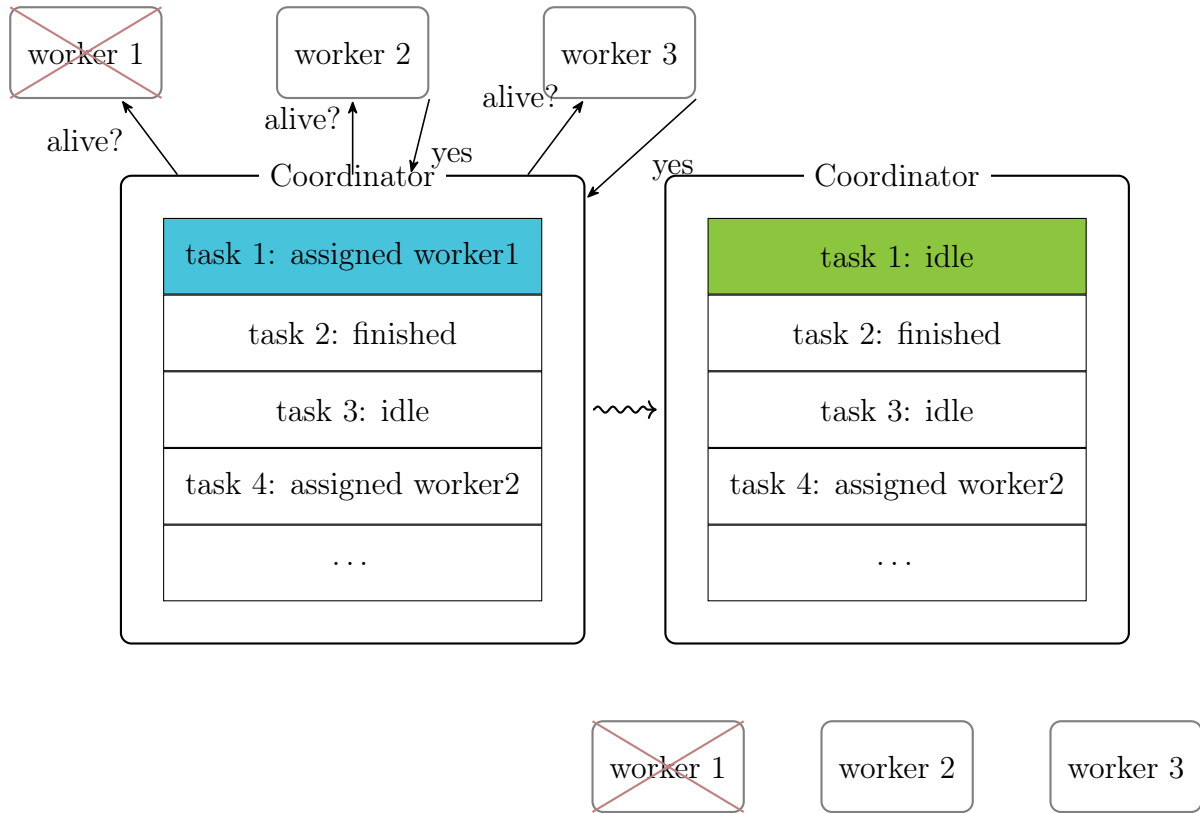


Figure 3.6: Coordinator handles failed workers.

However, one practical subtlety is that workers can fail at any point, even in the middle of writing the result to an intermediate file which is prepared for the next stage of evaluation. This can cause problems if other workers see these partial results. For example, if the worker crashes in the middle of writing a datalog atom `reachable(a, b)` to disk, then partial results such as “`reachab`” might be seen by other workers reading the results, which might confuse them. One quick way to solve this problem is to simply *ignore* such results. This is valid because the coordinator will reschedule such tasks later so other workers will complete them. Therefore the correct result `reachable(a,b)` *will* appear in a intermediate results file. Since we are working with a *set* of datalog atoms, duplications do not matter either. The correctness is therefore not affected.

Stragglers

As an extension, we consider another important abnormal behaviour: slow workers, or *stragglers*. There are many reasons why a worker can appear to be particularly slow in a distributed system, including stop-the-world garbage collection, operating system scheduling and bad disks [15, 43]. Stragglers can have a significant impact on the overall throughput of the engine [15, 57]. In the case of Erlog, due to the dependency of tasks across different stages, a worker has to wait for others to finish their tasks at the current

CHAPTER 3. IMPLEMENTATION

stage before going to the next stage. A slower worker can potentially become the bottleneck of the whole system if all other workers are waiting for its completion.

We simulate slow-down by asking some workers to sleep for some random period. This simulated behaviour is similar to a stop-the-world garbage collector. Whether or not this is a frequent behaviour depends on the actual environment, but it is one of many possible slow-down patterns in the real world.

One straightforward way of handling this is through timeout. The coordinator keeps track of how long each task has been running. If this exceeds some threshold, the coordinator will reschedule the task to another worker. The threshold can be determined by an Exponential Moving Average (EMA) [49], given in Equation (3.1). S_t is the value of EMA at time point t , and Y_t is the value to be averaged at time period t . In the case of Erlog, the coordinator will keep track of the time taken to complete each task and update the timeout threshold accordingly.

$$S_t = \begin{cases} Y_0 & t = 0 \\ \alpha Y_t + (1 - \alpha)S_{t-1} & t > 0 \end{cases} \quad (3.1)$$

The reason for choosing a moving average over a fixed threshold is that at each stage of computation, we may have inputs of different sizes, and a fixed threshold might not be able to characterise a suitable threshold for all stages.

However, as we shall see in the evaluation (§4.5.2), a coarse-grained EMA is not good enough to mitigate the effect of stragglers. This is because the throughput of the engine is limited by how fast and accurate it can detect stragglers. Even if there is only one slow worker, if the coordinator does not reassign its task to another worker, all workers would have to wait for it before going on to the next stage. This is especially important when the current stage is almost complete. Speculative execution [15] is a way of addressing this issue: launching currently in-progress tasks that are considered slow by the coordinator. This can be employed at a finer level to address the straggler issue further.

We wish to have more fine-grained information on the size of each task and the rate of each worker. The size of each task might vary depending on the actual graph, and the rate of each worker can help estimate how much time it will take for the worker to complete a task. If we were to launch in-progress tasks speculatively, we do not want to assign them to a slow worker which would make the whole process even slower.

To this end, we adopt a scheduler called the Longest Approximate Time to End (LATE) scheduler [57]. This scheduler is implemented to improve the performance of MapReduce applications in a heterogeneous environment. It offers a more fine-grained estimation of the working rate of each worker and hence the estimated completion time. The coordinator keeps track of the worker speeds in its `nodes_rate` field and updates this value according to the size of each task and the time taken to complete them. Based on this information, the coordinator can estimate whether a speculative task assigned to a worker will finish sooner than the current worker, and if so, we can launch such a task. Figure 3.7 shows the actions of the coordinator when it sees a slow task.

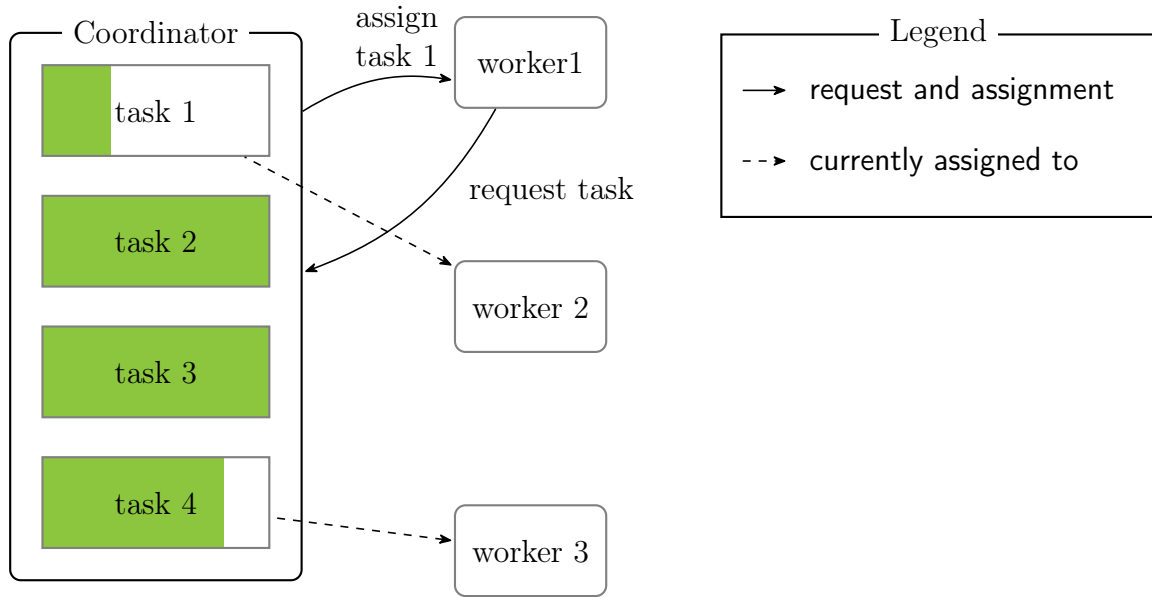


Figure 3.7: Worker 1 requests a task from the coordinator, and the coordinator finds that task 1 (assigned to worker 2 currently) would have been finished sooner if it were given to worker 1 now. Hence it assigns it to worker 1. Green bars indicate the progress of tasks.

Unreliable network

Erlang RPCs are not guaranteed to succeed [19]. In fact, networks can be unreliable, and messages might be arbitrarily delayed or even lost. Both the coordinator and workers should be able to handle these issues.

Erlog has a basic mechanism for handling delayed and lost messages. From the coordinator's point of view, duplicate worker requests might arrive due to retries. This is where `nodes_tasks` is used. When the coordinator receives a request for a task, but sees that the worker has already been assigned a task, the coordinator would just reassign the same task rather than looking for a new one.

Coordinator failures

The coordinator can fail in the real world as well. There are certainly ways to handle that. For example, we can have a Raft-replicated cluster or take periodic checkpoints of the coordinator. However, this is less likely to happen than worker failures since there is only one coordinator in the engine, therefore not worth the overhead of replication. If the coordinator fails, we can simply restart the task.

3.4.5 Worker jobs

Compared to the coordinator, workers have a much simpler workflow. A worker's main jobs are:

1. Request a task from the coordinator.
2. Read input from the disk (`tmp_path`) and carries out semi-naïve evaluation.

CHAPTER 3. IMPLEMENTATION

3. Hash partitions the result and persists it on disk.
4. Tell the coordinator that it has finished the task.
5. Repeat the above steps until told to stop by the coordinator.

Requesting tasks

The worker makes an `assign_task` RPC to the coordinator to request a task from it. The task record sent back from the coordinator contains information (the `tmp_path` field maintained by the coordinator) on where to find the input data. This includes both the accumulated database instance S^{i-1} and the delta instance Δ_S^i .

Again, RPCs are not guaranteed to succeed, so the worker will keep trying until the coordinator makes a response.

Semi-naïve evaluation

When the worker has read the input data, it will carry out one iteration of the semi-naïve evaluation. As a reminder from §3.3.4, the central formula for semi-naïve evaluation is

$$\Delta_S^{i+1} := P_S^i(I; T_l^{i-1}; T_l^i; \Delta_T^i) - S_i$$

Remember that with naïve evaluation, we take all the predicates from our accumulated database instance, whilst semi-naïve evaluation takes them from the Δ_T^i database instance. The worker does precisely that, and the Δ_T^i relation has been produced in the previous stage of computation and can be read off from one file.

However, the semi-naïve evaluation still needs access to the accumulated database instances. This can be explained through an example: imagine we have a rule that says

$$\text{anc}(x, y) \leftarrow \text{anc}(x, y), \text{anc}(z, y).$$

Then we need to generate two rules:

$$\begin{aligned} \text{temp}_{\text{anc}}^{i+1}(x, y) &\leftarrow \Delta_{\text{anc}}^i(x, z), \text{anc}(z, y) \\ \text{temp}_{\text{anc}}^{i+1}(x, y) &\leftarrow \text{anc}(x, z), \Delta_{\text{anc}}^i(z, y) \end{aligned}$$

Notice on the right hand side of the first rule, we have one Δ relation ($\Delta_{\text{anc}}^i(x, z)$), and one normal relation ($\text{anc}(z, y)$). The Δ relation is taken from the Δ database instance, but the normal relation is still taken from our accumulated full database instance.

Hash fragmentation & Finish tasks

When the worker has finished its evaluation, it would then hash partition its computed results to prepare for the next stage of evaluation.

When the task is finished, the worker sends an RPC to tell the coordinator. Again, we need to make sure that the coordinator receives the RPC through retries.

3.5 Implementing for evaluation

3.5.1 Benchmarking

The tools needed for evaluation are mainly based on Erlang’s benchmarking facilities, such as `timer:tc/3` [18]. It measures the wallclock time used by a program. There are also ways to measure CPU time, such as `statistics/1`. Wallclock time is chosen here because it considers the time spent in the kernel. This implementation involves many IO tasks, which we ought to consider when benchmarking. It is worth pointing out that although the documentation says it is wallclock time, `timer:tc/3` relies on `erlang:monotonic_time/0` to get the results, which is in fact Erlang VM’s monotonic clock [20], and this means that our results are not affected by clock drift.

However, one downside of measuring the wallclock time is that it is not stable, so experiments are run multiple times (generally around 10 times) to reduce the noise introduced by other activities of the kernel.

3.5.2 In-memory implementation

One disadvantage of the implementation is its intense IO work. The engine needs to persist results onto disk for all the intermediate results. To study the impact of this, an in-memory version of the distributed engine is also implemented. Most of the logic is the same, but there is no persistence of intermediate results. Instead, everything is message passed to the coordinator and stored in memory. We shall compare these two implementations in evaluation (§ 4.3).

3.6 Repository overview

Erlog uses the `rebar3` build tool [27], and the repository in Figure 3.8a is structured according to the OTP design principles [25], which specifies standards in terms of how processes, modules and directories should be laid out. The source code and tests mentioned below are all written by me.

3.6.1 Source code

Erlang has a flat module structure [55]. Therefore the source code is placed inside a single `src` folder. The main modules `coordinator` and `worker` are implemented using Erlang’s `gen_server` behaviour, and they correspond to the two different roles in the engine. They are set up as an Erlang cluster with Erlang nodes (virtual machines) and interact with each other through Erlang RPCs. The `eval` module implements the semi-naïve algorithm. Notice the supervision tree structure (§ 2.1.6) in Figure 3.8b, where `erlog_sup` invokes `coor_sup` and `worker_sup`, which then invokes `coordinator` and `worker` module. This follows Erlang’s “let it crash” design philosophy by building up a tree of supervision relation,

CHAPTER 3. IMPLEMENTATION

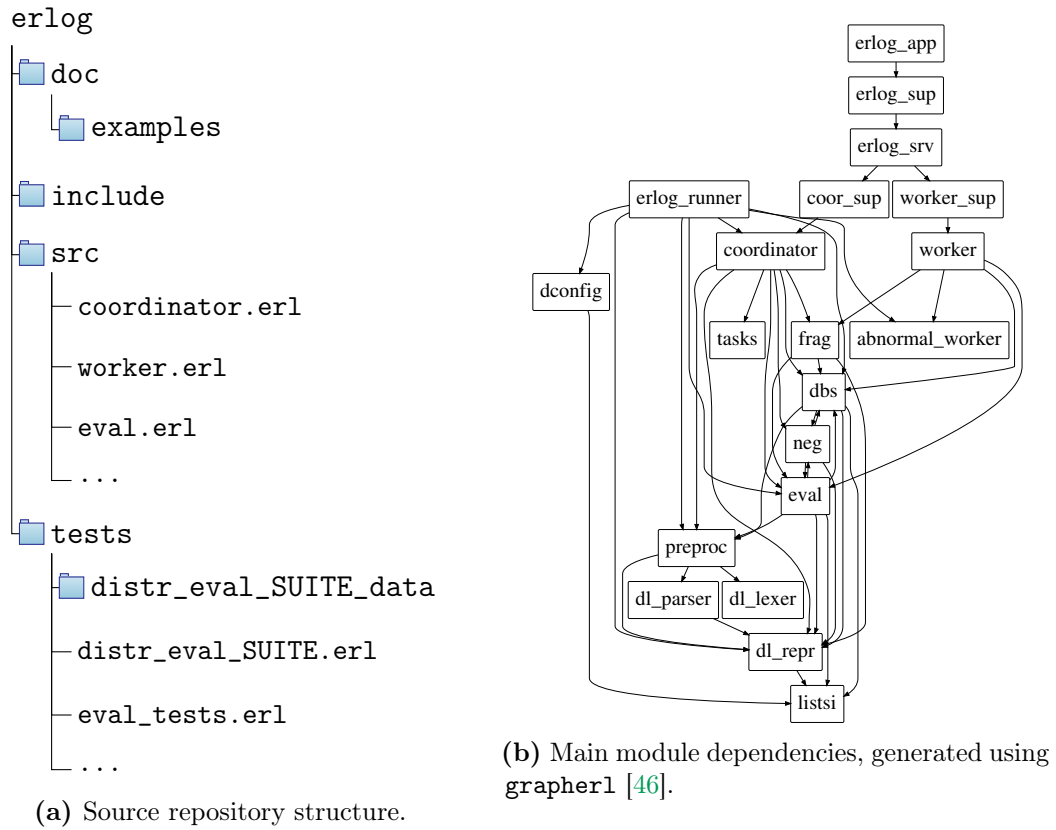


Figure 3.8: Directory structure and module dependencies of the project.

and in case one of the child processes crashes, its supervisor can attempt to restart it.

3.6.2 Tests

This project uses two Erlang testing frameworks: EUnit [21] for unit testing and Common Test [17] for larger-scale integration and end-to-end testing. They are stored in the testing directory. The convention for Common Test is to suffix `_SUITE` to the test file and `_SUITE_data` to the test data.

3.7 Summary

In this chapter, we looked at the implementation of Erlog. In particular, we designed the syntax of the language (§ 3.2). We also implemented single node evaluation using the semi-naïve algorithm (§ 3.3), which was then extended to the distributed evaluation algorithm (§ 3.4). We also looked at handling various aspects of fault tolerance while implementing the distributed algorithm (§ 3.4.4).

Chapter 4

Evaluation

This chapter evaluates the datalog engine against its success criterion. We shall first see that Erlog can correctly evaluate a Transitive Closure (TC) program (§ 4.2). After that, we show that the distributed engine performs better than the single node one (§ 4.3). Then we demonstrate the engine’s scalability in the number of workers (§ 4.4), followed by its tolerance against worker failures and scalability in the presence of them (§ 4.5).

4.1 Evaluation setup

The evaluation of this project is performed on a virtual machine. It has an AMD EPYC 7302 16 core 32 thread CPU, 64 GB memory and a 50 GB disk. It runs Ubuntu 20.04.3 LTS as its Operating System.

4.2 Acceptance testing

According to the success criteria of the project proposal ([Appendix D](#)), our engine should be able to evaluate the TC program on a graph of 100 nodes. The Soufflé benchmarking script [51] is adapted to generate a large graph with more than 100 nodes randomly. This correctness check is done using Erlang’s Common Test framework, and the test script tests the output produced by Erlog against the output of the Soufflé engine. Correctness on various programs (nine different programs) is checked, including TC (success criterion) and other variants such as strongly connected components, points-to analysis and negated programs (extension). The engine can indeed produce correct results on these programs.

4.3 Comparative evaluation

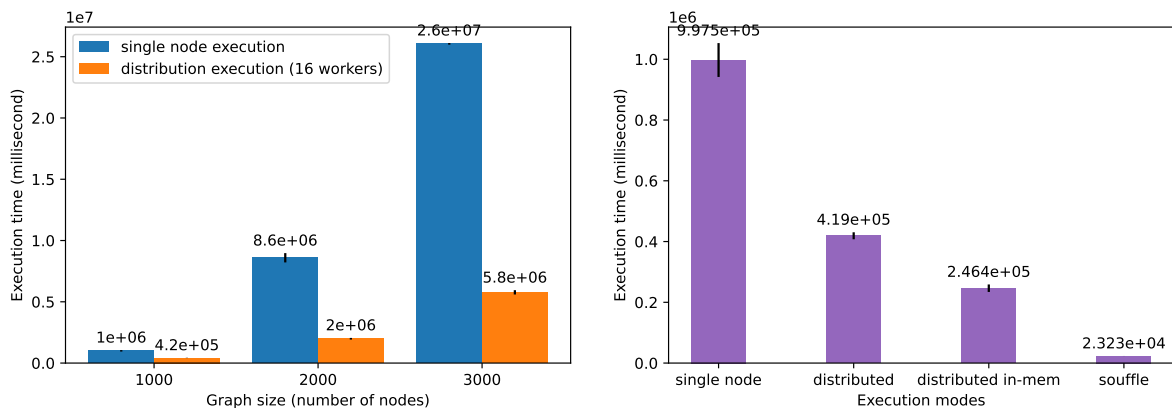
We start by comparing our distributed engine with the single node engine and see if we can achieve a lower latency in computing our TC program.

[Figure 4.1a](#) shows a comparison between the total time in computing the TC program on graph of {1000, 2000, 3000} nodes. We see that initially, we get about 2.5x speedup in terms of latency, and as we increase the size of the problem, we obtain even more speed up (about 4x–4.5x). This is because the larger the input is, the more we benefit from extracting its data parallelism. [Figure 4.1b](#) shows a more detailed comparison between different execution modes on a graph of 1000 nodes. The in-memory implementation of

CHAPTER 4. EVALUATION

the distributed engine achieves even better performance than the standard persistence version, albeit with a much higher memory cost.

This confirms our hypothesis in § 3.5.2: one of the bottlenecks of the previous implementation is its high IO cost. Figure 4.2a shows the flame graph [31] which breaks down the relative execution time spent in each function. In this graph we observe that the function `dbs:read_db/1` (`dbs:read_...` in the graph due to space limit), which mainly involves reading data from the disk, takes up a large proportion of the total working time. A second flame graph for the in-memory implementation is shown in Figure 4.2b. In this implementation, we have completely eliminated IO and are spending most of the time doing the actual semi-naïve evaluation (`eval:eval_seminaive_one/3` function).



(a) Comparison of two engines' execution times on graphs of different sizes. (b) Comparison of execution times of multiple modes and Soufflé.

Figure 4.1: Comparing the distributed engine with other execution modes.

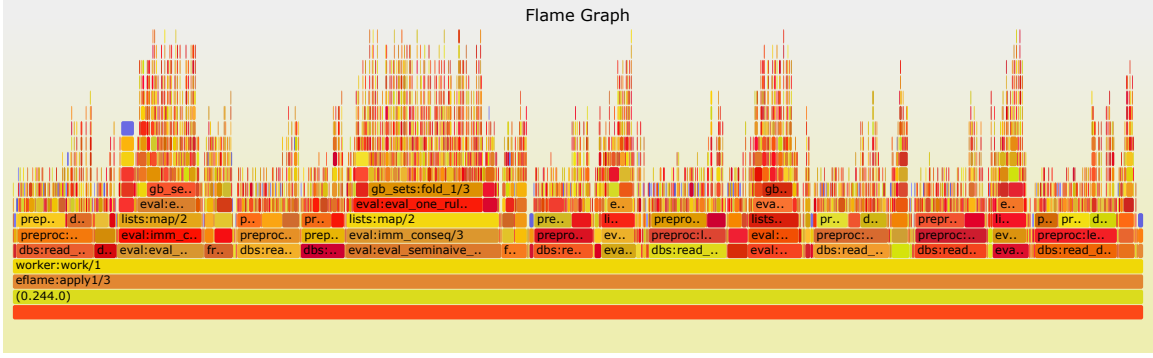
Thus to obtain higher performance, we would ideally want to keep the intermediate results in memory and choose to persist them on disk periodically for better fault tolerance. We shall discuss this further in future directions (§ 5.2).

4.4 Scalability

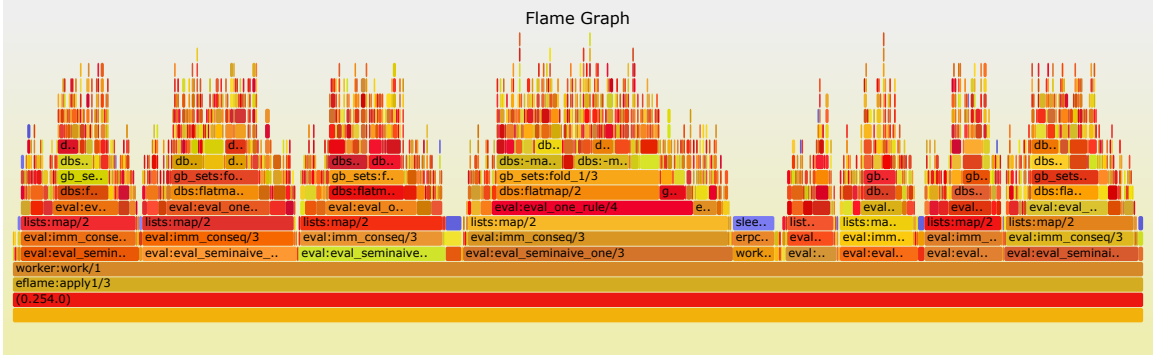
The evaluation of scalability can be thought of as addressing the question: *How can we add computing resources to handle the additional load?* [42]. Hence we want to evaluate how the engine's throughput changes as we add more and more workers to it.

We will consider different kinds of workloads, i.e. different kinds of programs. However, the nature of this implementation prevents us from evaluating how this engine responds to highly serial tasks such as Ackermann's function. This is because the advantage of this engine is data-parallel processing, so we need to be able to produce large input data to test the engine's ability. A program written to compute Ackermann's function does not fit such workflow and will not be considered.

In the following part, we are not going to use large graphs (tens of thousands of edges) as in the previous section, since they take a long time to compute (more than 6 hours for



(a) Flame graph for implementation with intermediate persistence.



(b) Flame graph for in-memory implementation.

Figure 4.2: Comparing the flame graph of two implementations. Generated using eflame [44].

one such computation). Instead, we will demonstrate the engine’s scalability by working with smaller graphs (thousands of edges).

4.4.1 Transitive Closure

Firstly, we evaluate the scalability of computing the TC program. The datalog program evaluated is listed in [Listing 2.1](#).

We carry out this experiment on two different types of graphs: 1. synthetic graphs, generated randomly using the Soufflé benchmarking script; 2. subsets of a real-world graph: Arxiv High-Energy Physics paper citation network [41]. Diagrams of throughput against the number of workers in both cases are given in [Figure 4.3](#). Throughput is calculated based on the ratio of graph size to the time taken to complete the computation, given by the following equation

$$\text{throughput} = k \frac{\text{graph size}}{\text{time}}$$

The unit of throughput is arbitrary and not meaningful. This benchmarking is done on input graphs of different sizes, and the graph size in the diagram represents the number of nodes. Each node has, on average, ten edges.

In both graphs, we can see a general trend of increasing throughput as we increase the number of workers in the engine, up to around 20 workers. The difference in throughput

CHAPTER 4. EVALUATION

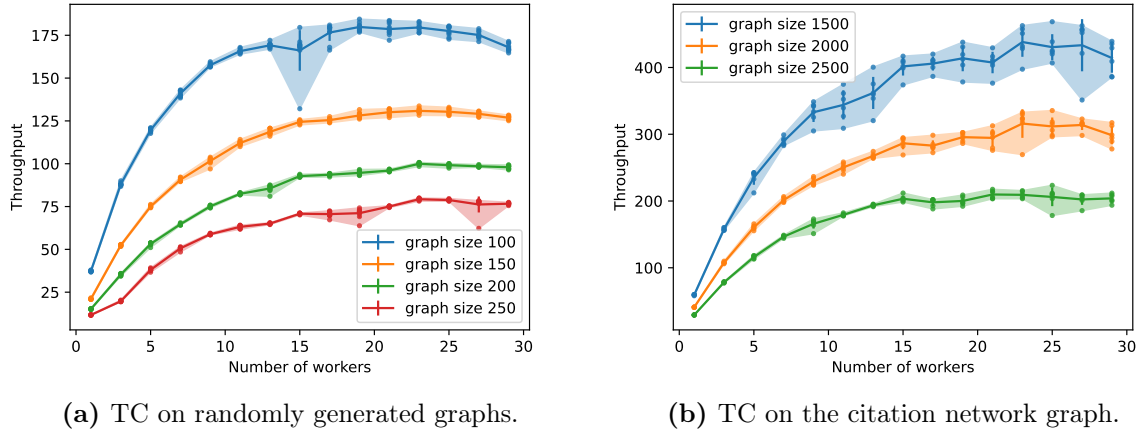


Figure 4.3: TC scalability on two different graphs.

between different graph sizes is due to the $\mathcal{O}(n^3)$ complexity to compute the TC: the increase in graph size does not catch up with the increase in the algorithm’s running time, hence the decreasing throughput on larger graphs. We observe there is a large variance in Figure 4.3a at 15 workers. This is likely due to garbage collection or other noise in the OS because this pattern does not emerge when I rerun the experiments with the same parameters.

4.4.2 Strongly Connected Component and Reverse Same Generation

A Strongly Connected Component (SCC) of a directed graph $G = (V, E)$ is a maximal set $C \subseteq V$ such that for every pair of vertices v and u in C , we have both $v \rightsquigarrow u$ and $u \rightsquigarrow v$ [14]. This can be done by computing the `reachable` predicate in TC twice. The Reverse Same Generation (RSG) program looks for nodes that have ancestors on the same “level”, which is also a similar workload to TC. Both programs are given in Listing 4.1 and Listing 4.2.

Since they are similar workloads, we would expect a similar shape from the scalability point of view, which is indeed the case, as shown in Figure 4.4.

```
reachable(X, Y) :- link(X, Y).
reachable(X, Y) :- link(X, Z),
    ↪ reachable(Z, Y).
scc(X, Y) :- reachable(X, Y),
    ↪ reachable(Y, X).
```

Listing 4.1: Strongly Connected Component program.

```
rsg(X, Y) :- flat(X, Y).
rsg(X, Y) :- up(X, X1), rsg(Y1, X1),
    ↪ down(Y1, Y).
```

Listing 4.2: Reverse Same Generation program.

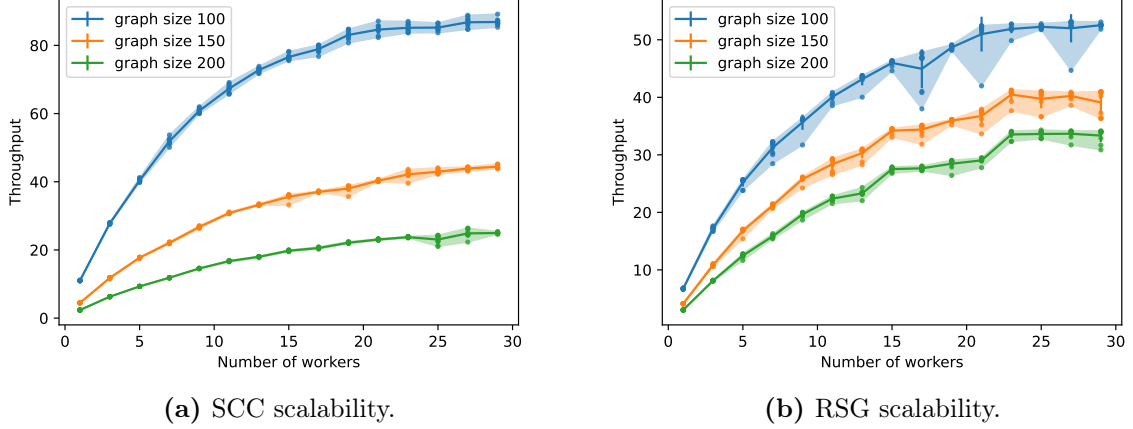


Figure 4.4: RSG and SCC scalability.

```

assign(Var1, Var2) :- primitiveAssign(Var1, Var2).
alias(InstanceVar, IVar) :- varPointsTo(InstanceVar, InstanceHeap), varPointsTo(IVar,
    ↪ InstanceHeap).
varPointsTo(Var, Heap) :- assignAlloc(Var, Heap).
varPointsTo(Var1, Heap) :- assign(Var2, Var1), varPointsTo(Var2, Heap).
assign(Var1, Var2) :- store(Var1, InstanceVar2, Field), alias(InstanceVar2,
    ↪ InstanceVar1), load(InstanceVar1, Var2, Field).

```

Listing 4.3: A simple context-insensitive, field-sensitive points-to analysis [51].

4.4.3 Points-to analysis

Points-to analysis aims to find the set of values that a pointer variable might point to, or in the case of a procedure, a description of what values it might read from or write to [39]. It is useful for the compiler to know this information at compile time to make transformations such as parallelising calls or load hoisting possible.

The program used for benchmarking is shown in Listing 4.3, and the throughput is shown in Figure 4.5. The decreasing throughput in the graph of size 50 is likely due to resource contentions. In a small-size problem, the frequency between a worker requesting a task and finishing it is short. This causes a large influx of RPCs on the coordinator, and in many cases, the coordinator will ask the workers to wait for others. This is a waste of time that could have been spent handling workers' messages to, for example, finish a task. However, this is usually not a problem if the input data is reasonably large, as the interval between a worker starting and finishing a task is relatively long. If the dataset is indeed small, we can simply choose a single-node datalog engine to do the job.

4.4.4 Experiments on AWS EC2 instances

As part of the extension, I performed experiments on a cluster of real distributed machines (AWS EC2 instances) and compared its scalability against running local Erlang nodes with Docker containers, with resource constraints to keep the comparison fair. Appendix B

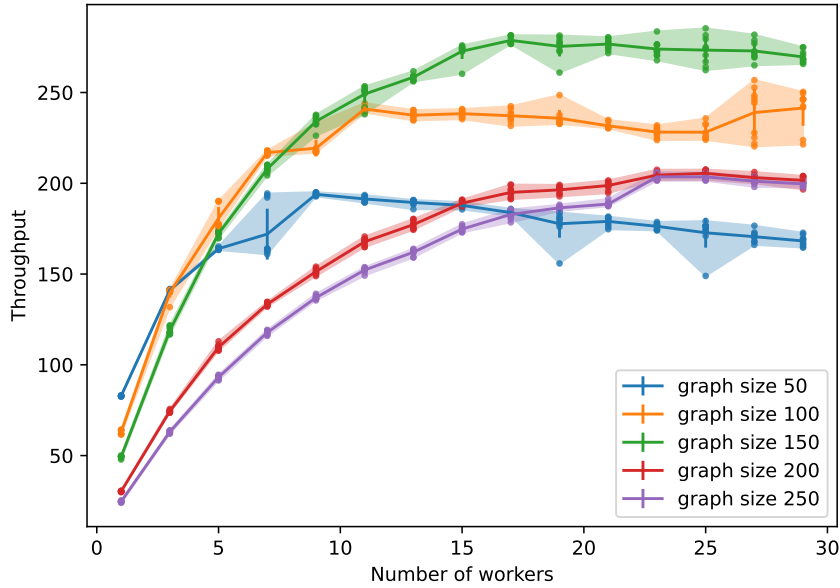


Figure 4.5: Points-to analysis program throughput.

contains detailed description of machine specifications. Figure 4.6 shows the results of such a comparison. Given that the CPU and memory are similar in these two cases, it is likely that the discrepancy comes from network throughput and latency.

Indeed, I measured network bandwidth^{*} and latency[†] for both EC2 instances and Docker container. The results are shown in Table 4.1. I have also measured the network throughput[‡] while the engine is computing the task in Figure 4.7, which shows that the network usage is far from the limit therefore it is likely that the latency is the limiting factor in this small-scale graph.

4.5 Fault tolerance

Fault tolerance is one of the central problems in distributed systems. Therefore it is worthwhile to evaluate how this engine responds to faults. This section looks at how abnormally-behaving workers can impact the engine. In particular, we focus on how correctness (§4.5.1) and scalability (§4.5.2) are affected. The supervision feature of Erlang (§2.1.6) is turned off during evaluation since we wish to see the impact of crash-stop failures.

^{*}measured using `iperf3 -c <ip-addr> -p <port-num>`

[†]measured using `ping <ip-addr>`

[‡]measured using cAdvisor: <https://github.com/google/cadvisor>, retrieved on 2022-04-27

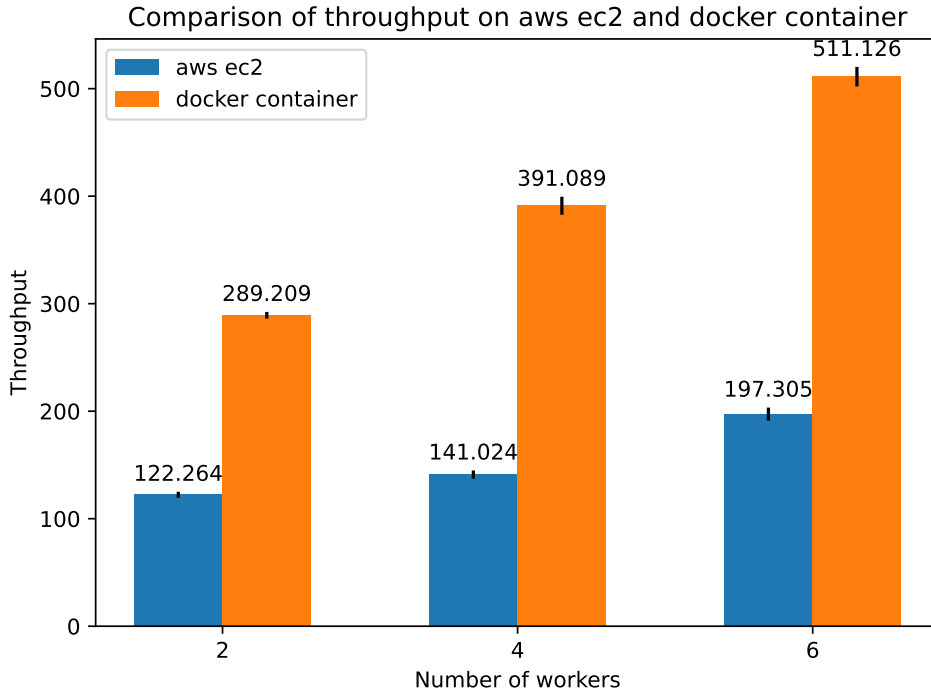


Figure 4.6: AWS and local Docker container throughput comparison.

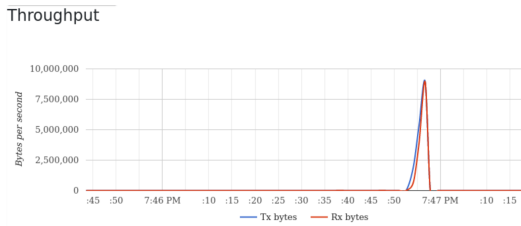


Figure 4.7: Network throughput of containers during computation.

	throughput	latency
EC2	994 Mbits/sec	0.409 ms
Docker	19.7 Gbits/sec	0.0612 ms

Table 4.1: Comparison of network latency and throughput between EC2 and local Docker containers.

4.5.1 Correctness

To demonstrate that the engine indeed produces correct results in the presence of worker failures, we can perform experiments to see whether different kinds of abnormal behaviours will affect the correctness of the output. This is done by killing or slowing down workers at random points, and writing additional tests to model these behaviours. These tests are run multiple times (each test repeated five times) with different percentages of faults to ensure sufficient coverage. The engine’s output is, again, checked against results produced by the Soufflé engine. The results show that the engine can output correct results even if some workers crash or are running slowly.

4.5.2 Throughput

Knowing that our engine produces correct output in the presence of worker failures, we can look at how worker faults might affect the throughput of our engine. As mentioned

CHAPTER 4. EVALUATION

in §3.4.4, we are going to consider two types of behaviours: crash-stop workers and slow workers. We look at each of them respectively in the following two sections.

Crash stop workers

We first look at how failed workers affect the throughput of the engine. Here we are going to focus on the TC program again. A medium-sized graph is chosen with about 150 nodes and 1500 edges, and this experiment is repeated for different failure percentages. The results are shown in Figure 4.8. At failure rate 0.2, we see that the throughput is relatively consistent, and as we have more and more worker failures, the variance increases. This is expected since the coordinator reschedules the task when the worker died, and the exact time of this is non-deterministic. For example, if a worker crashes when it is about to finish its job, then this would be a much worse situation than the worker crashing when it just started its computation.

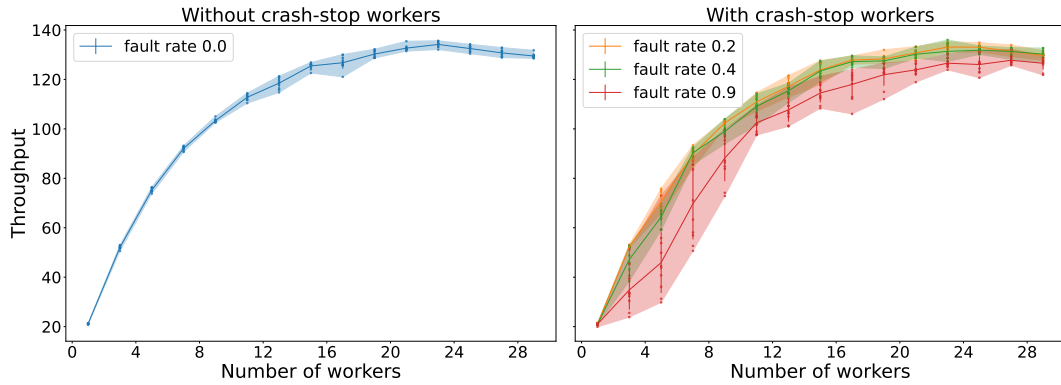


Figure 4.8: TC program throughput in the presence of worker failures. Failure rate is defined as the percentage of workers that will fail.

Stragglers

In this section, we compare three strategies for dealing with slow workers: 1. Do nothing 2. Use an average completion time 3. Use a fine-grained estimation of completion time.

If we do nothing to handle slow workers, then the execution time will be dominated by stragglers. This means that the engine is as slow as the slowest worker in the system, i.e. one worker can slow down the engine however much it wants to. This would usually cause the engine to time out. Although if we wait long enough, the engine can still give us the final results.

If we only use the EMA timeout mechanism, then we can see that in the left part of Figure 4.9, the throughput of the engine reaches a bottleneck even though we keep adding more workers to it. This is because an aggregated completion time is not good enough since the average time is also affected by slow workers. Moreover, the hash partition of our data is not balanced, which adds heterogeneity to our system. This implies that we cannot use a fixed value as the (estimated) total work for each worker.

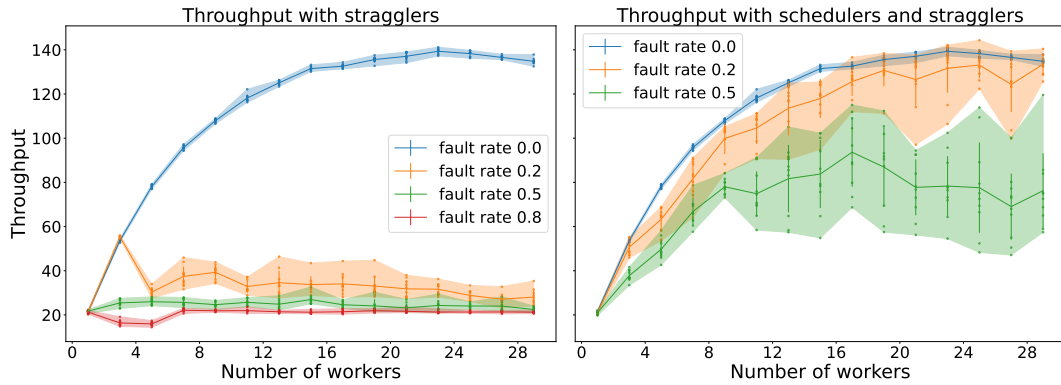


Figure 4.9: TC program throughput when there are stragglers. The engine maintains scalability with the help of a scheduler.

To address this issue, I developed a more fine-grained scheduling algorithm (§ 3.4.4). The right half of Figure 4.9 shows the effect of doing such an estimation on stragglers. The throughput variance is still large, which is as expected since the estimation cannot be perfectly accurate, plus the timing of workers getting slower is non-deterministic.

We have already discussed the implementation of our scheduler: it keeps track of an approximated working rate of each worker and takes into account the size of each task. Moreover, it only reassigns an in-progress task if the estimated completion time of the new worker is *shorter* than the rest of the time needed by the current worker. In other words, the scheduler will not reassign a task if the requesting worker needs a longer time to complete that in-progress task. This level of detailed estimation helps us address the heterogeneity and more accurately identify tasks that should be speculatively launched. As a result, we mitigate the problem caused by stragglers.

Dealing with slow workers itself is a complex scheduling problem which comes up in many areas of Computer Science, such as operating system scheduling [8], as well as data centre resource sharing [37]. The scheduler implemented in this project is only a first attempt and might not be able to deal with other patterns of slowness. We shall return to this in the future work (§ 5.2) section of the next chapter.

4.6 Summary

In this chapter, we evaluated the Erlog engine. We performed acceptance testing to show that the engine can compute Transitive Closure correctly in a distributed manner (§ 4.2), even with worker failures (§ 4.5.1). We also showed that the engine scales well (with up to 6.5x throughput in our evaluation setup), both in a normal environment (§ 4.4) and in the presence of failed and slow workers (§ 4.5), by scheduling tasks according to the rate of each worker.

Chapter 5

Conclusions

In this dissertation, we looked at a distributed datalog engine, *Erlog*, that exploits data-parallelism while evaluating datalog programs. It utilises a *coordinator-worker* approach and takes advantage of the parallelism in the RA primitives, via which datalog queries are implemented.

5.1 Achievements

In the introduction, I talked about the emergence of large amounts of data stored in database systems, requiring a highly parallel query system. We also observed the coarse-grain property of RA operations, suggesting a distributed way of evaluating a datalog program.

Erlog demonstrates strong scalability on a multi-core machine (§ 4.4), with up to 6.5x throughput increase. It is also resilient against worker failures (§ 4.5). I have shown that the correctness of the engine is not affected by crash-stop workers (§ 4.5.1), moreover, with the help of scheduling algorithms (extension), the engine can also maintain its scalability in the presence of stragglers (§ 4.5.2).

Overall the project was successful. I developed the front end of the datalog engine (§ 3.2) to conveniently parse an input program. I also implemented the semi-naïve algorithm for datalog program evaluation on a single node (§ 3.3). Then I extended the serial engine into a distributed one using a hash-based partition and join approach, in a coordinator-worker paradigm (§ 3.4). Moreover, I incorporated negation into the core datalog program as an extension (§ 3.3.5), which allows more parallelism across different rules of one program. The engine can evaluate the transitive closure program on a graph with more than 100 nodes (§ 4.2), as required by the success criterion. In fact, it has exceeded this criterion as it can handle larger graphs and additional programs such as points-to analysis (§ 4.4.3).

5.2 Future work

While evaluating the engine, I have discovered several areas in which improvements can be made:

- In § 4.3, we saw the benefit of having in-memory computation as opposed to relying on file IO at each stage. This is due to the fact that datalog evaluation requires lots of data sharing in between stages. To address this, we could use a good abstraction

of shared distributed memory, which is indeed provided by RDD (§ 2.1.5). This has the additional benefit of sharing memory between multiple workers, reducing the overhead of duplication.

- This engine implements the simple distributed join algorithm (§ 3.4.2). However, more advanced distributed join algorithms are available, such as radix hash and sort-merge hash [9]. They may exploit more hardware optimisation.
- The scheduling algorithm dealing with stragglers is by no means comprehensive. More complex scheduling algorithms, perhaps a more complete implementation of the LATE scheduler [57] can be developed to handle more heterogeneity, although this depends on the environment in which the engine is running.

5.3 Lessons learned

This project is my first time using Erlang to develop a large project on distributed systems. As with most distributed systems, the devil is in the details. It is notoriously hard to debug a distributed system due to the non-deterministic and complex interactions between nodes. Being systematic and patient in writing and reading log messages was an effective technique.

Although fault tolerance of the engine was guaranteed by the framework used, there were still many subtle issues that I needed to consider while implementing it. It is often prohibitively expensive to remove all of the problems in a distributed system. This helps me appreciate Erlang’s “let it crash” philosophy. Learning a new language and, consequently, new ways of solving problems have always been enjoyable to me. I am glad this project has provided this opportunity.

Bibliography

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1995. ISBN 0201537710.
- [2] D. Anderson. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010. ISBN 9780984521401. URL <https://books.google.co.uk/books?id=RJ0VUkfUWZkC>.
- [3] A. Andersson. General balanced trees. *Journal of Algorithms*, 30(1):1–18, 1999. ISSN 0196-6774. doi: <https://doi.org/10.1006/jagm.1998.0967>. URL <https://www.sciencedirect.com/science/article/pii/S0196677498909671>.
- [4] Apache Software Foundation. Apache license, Jan 2004. URL <https://www.apache.org/licenses/LICENSE-2.0>.
- [5] Apache Software Foundation. Spark: Unified engine for large-scale data analytics, 11 May 2022. URL <https://spark.apache.org>.
- [6] J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, sep 2010. ISSN 0001-0782. doi: [10.1145/1810891.1810910](https://doi.org/10.1145/1810891.1810910). URL <https://doi.org/10.1145/1810891.1810910>.
- [7] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo. The deductive database system *ldl++*. *Theory Pract. Log. Program.*, 3(1):61–94, jan 2003. ISSN 1471-0684. doi: [10.1017/S1471068402001515](https://doi.org/10.1017/S1471068402001515). URL <https://doi.org/10.1017/S1471068402001515>.
- [8] J. Bacon and T. Harris. *Operating Systems, Concurrent and Distributed Software Design*. Addison Wesley, 2003. ISBN 0-321-11789-1.
- [9] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoeffler. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10(5):517–528, jan 2017. ISSN 2150-8097. doi: [10.14778/3055540.3055545](https://doi.org/10.14778/3055540.3055545). URL <https://doi.org/10.14778/3055540.3055545>.
- [10] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988. doi: [10.1109/2.59](https://doi.org/10.1109/2.59).
- [11] F. Cacace and M. Houtsma. An overview of parallel strategies for transitive closure on algebraic machines. volume 503, pages 44–62, 09 1990. ISBN 978-3-540-54132-5. doi: [10.1007/3-540-54132-2_49](https://doi.org/10.1007/3-540-54132-2_49).
- [12] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan. Flumejava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference*

- on *Programming Language Design and Implementation (PLDI)*, pages 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010. URL <http://dl.acm.org/citation.cfm?id=1806638>.
- [13] B. Chin, D. von Dincklage, V. Ercegovac, P. Hawkins, M. S. Miller, F. Och, C. Olston, and F. Pereira. Yedalog: Exploring knowledge at scale. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, pages 63–78, Dagstuhl, Germany, 2015. URL http://drops.dagstuhl.de/opus/frontdoor.php?source_opus=5017.
 - [14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844.
 - [15] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI’04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.
 - [16] I. S. e Souza. erlgrind., 3 May 2022. URL <https://github.com/isacssouza/erlgrind>.
 - [17] Ericsson AB. Common test user’s guide, 22 Mar 2022. URL https://www.erlang.org/doc/apps/common_test/users_guide.html. Version 1.22.1.
 - [18] Ericsson AB. Erlang efficiency guide advanced, 22 Mar 2022. URL https://www.erlang.org/doc/efficiency_guide/advanced.html. Version 1.22.1.
 - [19] Ericsson AB. Enhanced remote procedure call module documentation, 22 Mar 2022. URL <https://www.erlang.org/doc/man/erpc.html>.
 - [20] Ericsson AB. Erlang run-time system application (erts), 02 May 2022. URL https://www.erlang.org/doc/apps/erts/time_correction.html.
 - [21] Ericsson AB. Eunit user’s guide, 22 Mar 2022. URL https://www.erlang.org/doc/apps/eunit/users_guide.html. Version 2.7.
 - [22] Ericsson AB. fprof, 03 May 2022. URL <https://www.erlang.org/doc/man/fprof.html>.
 - [23] Ericsson AB. Erlang leex, 22 Mar 2022. URL <https://www.erlang.org/doc/man/leex.html>.
 - [24] Ericsson AB. Erlang yecc, 22 Mar 2022. URL <https://www.erlang.org/doc/man/yecc.html>.
 - [25] Ericsson AB. Otp design principles user’s guide, 22 Mar 2022. URL https://www.erlang.org/doc/design_principles/users_guide.html.
 - [26] Erlang Community. Erlang, 22 Mar 2022. URL <https://www.erlang.org/about>.

BIBLIOGRAPHY

- [27] Erlang Community. Rebar3, 22 Mar 2022. URL <http://rebar3.org>.
- [28] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, USA, 2 edition, 2008. ISBN 9780131873254.
- [29] T. Gilray, S. Kumar, and K. Micinski. Compiling data-parallel datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 23–35, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383257. doi: 10.1145/3446804.3446855. URL <https://doi.org/10.1145/3446804.3446855>.
- [30] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. *Datalog and Recursive Query Processing*. Now Foundations and Trends, 2013.
- [31] B. Gregg. Visualizing performance with flame graphs. Santa Clara, CA, July 2017. USENIX Association.
- [32] T. Griffin. Part ia databases lecture notes, Oct 2021. URL https://www.cl.cam.ac.uk/teaching/2122/Databases/databases_2021.pdf.
- [33] W. G. Griswold and G. M. Townsend. The design and implementation of dynamic hashing for sets and tables in icon. *Softw. Pract. Exper.*, 23(4):351–367, apr 1993. ISSN 0038-0644.
- [34] B. Gustavsson. Sets, ordsets, gb_sets, sofs. what is going on?, 26 Mar 2010. URL <http://erlang.org/pipermail/erlang-questions/2010-March/050332.html>.
- [35] F. Hebert. *Stuff Goes Bad, Erlang in Anger*. Online, June 2016.
- [36] J. M. Hellerstein and M. Stonebraker. *Readings in Database Systems: Fourth Edition*. The MIT Press, 2005. ISBN 0262693143.
- [37] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI’11, page 295–308, USA, 2011. USENIX Association.
- [38] Infor. Logicblox – next generation analytics applications, Apr 2022. URL <https://developer.logicblox.com>.
- [39] T. Jones. Part ii optimising compilers, 2022.
- [40] H. Jordan, P. Subotić, D. Zhao, and B. Scholz. A specialized b-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPOPP ’19, page 327–339, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362252. doi: 10.1145/3293883.3295719. URL <https://doi.org/10.1145/3293883.3295719>.

- [41] KDD Cup. Citation graph of the hep-th portion of the arxiv, 2003.
- [42] M. Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, Beijing, 2017. ISBN 978-1-4493-7332-0. URL <https://www.safaribooksonline.com/library/view/designing-data-intensive-applications/9781491903063/>.
- [43] M. Kleppmann. Part ib concurrent and distributed systems lectures, 2020.
- [44] V. Kyrilov. Flame graphs for erlang., 2 May 2022. URL <https://github.com/proger/eflame>.
- [45] M. E. Lesk and S. C. J. E. Schmidt. The lex & yacc page, 27 Mar 2022. URL <http://dinosaur.compilertools.net>.
- [46] A. Lindberg. Grapherl: Create graphs of erlang systems and programs., 26 Mar 2022. URL <https://github.com/eproxus/grapherl>.
- [47] M. Madsen and O. Lhoták. Fixpoints for the masses: Programming with first-class datalog constraints. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi: 10.1145/3428193. URL <https://doi.org/10.1145/3428193>.
- [48] S. Moore and T. Jones. Part ib computer design lectures, 2020.
- [49] NIST. *NIST/SEMATECH e-Handbook of Statistical Methods*. 2013. URL <https://doi.org/10.18434/M32189>.
- [50] R. Ramakrishnan, W. G. Roth, P. Seshadri, D. Srivastava, and S. Sudarshan. The coral deductive database system. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 544–545, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915925. doi: 10.1145/170035.171550. URL <https://doi.org/10.1145/170035.171550>.
- [51] Soufflé developers. Soufflé benchmarks, 2021. URL <https://github.com/souffle-lang/benchmarks.git>.
- [52] Soufflé developers. Soufflé: A datalog synthesis tool for static analysis, Apr 2022. URL <https://souffle-lang.github.io>.
- [53] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285 – 309, 1955. doi: pjmath/1103044538. URL <https://doi.org/>.
- [54] E. V.-M. Tristan Sloughter, Fred Hebert. *Adopting Erlang*. Online, August 2019.
- [55] F. Trottier-Hebert. *Learn You Some Erlang for Great Good!* online, 2013. URL <https://learnyousomeerlang.com>.
- [56] J. Weidendorfer. kcachegrind, callgraphviwer, 3 May 2022. URL <https://kcachegrind.github.io/html/Home.html>.

BIBLIOGRAPHY

- [57] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 29–42, USA, 2008. USENIX Association.
- [58] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, Apr. 2012. USENIX Association. ISBN 978-931971-92-8. URL <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>.

Appendix A

Coordinator-Worker interaction

A.1 Coordinator server RPC handling

Listing A.1 are example calls provided by the coordinator that can assign a task to the requesting worker and mark a task as finished.

```
handle_call(stage_num, _From, State = #coor_state{stage_num = StageNum}) ->
  {reply, StageNum, State};
handle_call(tmp_path, _From, State = #coor_state{tmp_path = TmpPath}) ->
  {reply, TmpPath, State};
handle_call(prog_num, _From, State = #coor_state{prog_num = ProgNum}) ->
  {reply, ProgNum, State};
handle_call(num_tasks, _From, State = #coor_state{num_tasks = NumTasks}) ->
  {reply, NumTasks, State};
handle_call({assign, WorkerNode},
  _From,
  State = #coor_state{tasks = Tasks, nodes_tasks = NodesTasks}) ->
  case tasks:is_eval(
    maps:get(WorkerNode, NodesTasks))
  of
    true -> % if assigned an eval task, then give it a wait one
      {reply, maps:get(WorkerNode, NodesTasks), State};
    false ->
      {Task, NewState} = find_next_task(State, WorkerNode),
      ?LOG_DEBUG("#{assigned_task_from_server => Task, to => WorkerNode}),
      ?LOG_DEBUG("#{old_tasks => Tasks, new_tasks => NewState#coor_state.tasks}),
      {reply, Task, NewState#coor_state{nodes_tasks = NodesTasks#{WorkerNode =>
        ↪ Task}}}}
  end;
handle_call({finish, Task, WorkerNode},
  _From,
  State = #coor_state{nodes_tasks = NodesTasks}) ->
  NewState = update_finished_task(Task, State),
  {reply,
    ok,
    NewState#coor_state{nodes_tasks = NodesTasks#{WorkerNode =>
      ↪ tasks:new_wait_task()}}};
handle_call(finished, _From, State = #coor_state{tasks = Tasks}) ->
  case Tasks of
    [#task{type = terminate}] ->
      {reply, true, State};
```

APPENDIX A. COORDINATOR-WORKER INTERACTION

```
_Ts ->
    {reply, false, State}
end;
handle_call(terminate, _From, State) ->
    {stop, normal, ok, State}.

handle_cast({reg, WorkerNode},
    State = #coord_state{nodes_rate = NodesRate, nodes_tasks = NodesTasks}) ->
    NewNodesRate = NodesRate#{WorkerNode => ?INITIAL_RATE}, % to avoid division by zero
    NewNodesTasks = NodesTasks#{WorkerNode => tasks:new_wait_task()},
    {noreply, State#coord_state{nodes_rate = NewNodesRate, nodes_tasks = NewNodesTasks}}.
```

Listing A.1: Coordinator handlers for RPCs.

A.2 Worker task computation

Listing A.2 shows that the worker requests a task from the coordinator, finds the appropriate input data, and carries out computation. Once it has finished its computation, it would then persist the results before calling the `finish_task` RPC.

```
#worker_state{num_tasks = NumTasks,
    tmp_path = TmpPath,
    mode = Mode}) ->
case call_coor(assign_task, [node()]) of
    T = #task{type = evaluate,
        task_num = TaskNum,
        stage_num = StageNum,
        prog = Program,
        prog_num = ProgNum} ->
    % HACK no need to distinguish different stages for the FullDB, just keep adding
    FName1 = io_lib:format("~sfulldb-~w-~w-~w", [TmpPath, ProgNum, 1, TaskNum]),
    % here we are combining all full_dbs together, so when we generate
    % new atoms at each worker, we can avoid duplicates being written to the
    % deltas. The need for this is due to the fact that one atom that has been
    % written to the delta and indeed used in previous iterations are not
    ↪ necessarily
    % from that worker, but there is no way for the generating worker to know
    % that atom has already been generated.
    FullDBPath = io_lib:format("~s-~w", [TmpPath ++ "fulldb", ProgNum]),
    FullDB = dbs:read_db(FullDBPath),

    check_straggle(Mode),
    % we need the old delta db because we might want the delta db generated from
    % other workers
    lager:debug("worker_node ~p, reading_fullldb_from_file_named ~p", [node(),
    ↪ FName1]),
    FName2 = io_lib:format("~stask-~w-~w-~w", [TmpPath, ProgNum, StageNum,
    ↪ TaskNum]),
```

A.2. WORKER TASK COMPUTATION

```
% we need to take the diff between this delta and the FullDB because our delta
% might be generated by other workers
DeltaDB = dbs:read_db(FName2),
lager:debug("reading_deltas_from_file_named ~p, delta_db_read_size ~p",
            [FName2, dbs:size(DeltaDB)]),
% use imm_conseq/3
% need to store FullDB somewhere for later stages of evaluation
% and this is not a static state, it changes every iteration
% and is potentially huge, so this cost might be quite large
{NewFullDB, NewDeltaDB} = eval:eval_seminative_one(Program, FullDB, DeltaDB),
lager:debug("new_db ~p", [dbs:to_string(NewDeltaDB)]),
% hash the new DB locally and write to disk
% with only tuples that have not been generated before
frag:hash_frag(NewDeltaDB, Program, ProgNum, StageNum + 1, NumTasks, TmpPath ++
    ↪ "task"),
FullDBToWrite =
    dbs:subtract(
        dbs:union(NewFullDB, DeltaDB), FullDB),
dbs:write_db(FullDBPath, FullDBToWrite),
% call finish task on coordinator
finish_task(T),
lager:debug("~p rpc results for finish at stage ~p task ~p", [node(), StageNum,
    ↪ TaskNum]),
% request new tasks
lager:debug("~p stage ~w task ~w finished, requesting new task",
            [node(), StageNum, TaskNum]),
NewState = State#worker_state{task_num = TaskNum, stage_num = StageNum},
lager:debug("worker_node ~p, new_state ~p", [node(), NewState]),
work(NewState);
#task{type = wait} ->
lager:debug("~p this is a wait task, sleeping for ~p sec", [node(), ?SLEEP_TIME
    ↪ / 1000]),
timer:sleep(?SLEEP_TIME),
work(State);
#task{type = terminate} ->
lager:debug("~p all done, time to relax", [node()]);
Other ->
lager:info("~p some other stuff ~p~n", [node(), Other])
end.
```

Listing A.2: Worker requesting tasks from coordinator and does computation.

Appendix B

AWS and Docker Specification

B.1 AWS EC2 specification

Table B.1 contains the specification of the EC2 instances that I used for evaluating the datalog engine.

role	instance	vCPU	Mem (GiB)	Storage	Network Performance
coordinator	t3.small	2	2	AWS EBS ^a + EFS ^b	Up to 5 Gbps
worker	t2.micro	1	1	AWS EBS + EFS	Low to Moderate

^a For local data storage, <https://aws.amazon.com/ebs/>, retrieved 2022-04-27

^b For shared storage, <https://aws.amazon.com/efs/>, retrieved 2022-4-27

Table B.1: AWS instances specification

B.2 Docker container configuration

Listing B.1 lists the docker compose file, notice the resource limits that are used in order to keep the CPU and memory resource the same as EC2 instances.

```
version: "3.7"
services:
  coor:
    image: erlog
    deploy:
      resources:
        limits:
          cpus: '2.0'
          memory: 2G
    command: rebar3 shell --name 'coor@host.com' --setcookie erlog_cookie
    container_name: host.com
    working_dir: /app
    volumes:
      - ./:/app
    tty: true
    networks:
      - net1
```

B.2. DOCKER CONTAINER CONFIGURATION

```
w1:
  image: erlog
  deploy:
    resources:
      limits:
        cpus: '1.0'
        memory: 1G
    command: rebar3 shell --name 'worker1@host1.com' --setcookie erlog_cookie
    container_name: host1.com
    working_dir: /app
    volumes:
      - ./:/app
    depends_on:
      - coor
    tty: true
    networks:
      - net1

w2:
  image: erlog
  deploy:
    resources:
      limits:
        cpus: '1.0'
        memory: 1G
    command: rebar3 shell --name 'worker2@host2.com' --setcookie erlog_cookie
    container_name: host2.com
    working_dir: /app
    volumes:
      - ./:/app
    depends_on:
      - coor
    tty: true
    networks:
      - net1

w3:
  image: erlog
  deploy:
    resources:
      limits:
        cpus: '1.0'
        memory: 1G
    command: rebar3 shell --name 'worker3@host3.com' --setcookie erlog_cookie
    container_name: host3.com
    working_dir: /app
    volumes:
      - ./:/app
    depends_on:
      - coor
```

APPENDIX B. AWS AND DOCKER SPECIFICATION

```
  tty: true
  networks:
    - net1

w4:
  image: erlog
  deploy:
    resources:
      limits:
        cpus: '1.0'
        memory: 1G
  command: rebar3 shell --name 'worker4@host4.com' --setcookie erlog_cookie
  container_name: host4.com
  working_dir: /app
  volumes:
    - .:/app
  depends_on:
    - coor
  tty: true
  networks:
    - net1

w5:
  image: erlog
  deploy:
    resources:
      limits:
        cpus: '1.0'
        memory: 1G
  command: rebar3 shell --name 'worker5@host5.com' --setcookie erlog_cookie
  container_name: host5.com
  working_dir: /app
  volumes:
    - .:/app
  depends_on:
    - coor
  tty: true
  networks:
    - net1

w6:
  image: erlog
  deploy:
    resources:
      limits:
        cpus: '1.0'
        memory: 1G
  command: rebar3 shell --name 'worker6@host6.com' --setcookie erlog_cookie
  container_name: host6.com
  working_dir: /app
```

B.2. DOCKER CONTAINER CONFIGURATION

```
volumes:
  - ./:/app
depends_on:
  - coor
tty: true
networks:
  - net1

networks:
  net1:
    driver: bridge
```

Listing B.1: Docker compose file.

Appendix C

Software libraries versions

C.1 Software tools used in this project

Table C.1 shows the software versions and license used in this project.

Library name	Version	License
Erlang/OTP	24.1	Apache-2.0
rebar3	3.18.0	Apache-2.0
recon	2.5.2	BSD 3-Clause
lager	3.9.2	Apache-2.0
grapherl	2c5d0e4	Apache-2.0
erlang_ls	0.28.0	Apache-2.0
elvis	1.1.0	Apache-2.0
cAdvisor	0.39.3	Apache-2.0
eflame	1.0.1	ISC
erlgrind	357a47e	BSD
KCacheGrind	0.8.0	GPL V2

Table C.1: Software library versions in this project.

Appendix D

Project Proposal

The original proposal can be found on the next page.

Computer Science Tripos – Part II Project Proposal

Erlog: Distributed Datalog Engine

Candidate 2030B

17th Oct, 2021

Project Originator: Mistral Contrastin

Project Supervisors: Mistral Contrastin and Dr A. Madhavapeddy

Director of Studies: Dr J. K. Fawcett

Project Overseers: Dr R. Mullins and Prof. M. Fiore

1 Introduction and Description

This project aims to implement a distributed Datalog engine that can distribute the work of evaluating a Datalog program to multiple workers that are executing in parallel.

Datalog is rooted in the database systems community and was first developed in the eighties and early nineties where some early systems were built by academia such as Coral and LDL++. It has since entered a long dormancy due to the lack of compelling applications. In recent years, however, there has been an increase in the use of recursive query languages to do information extraction, program analysis, declarative networking, etc. Datalog has regained its popularity due to these emerging new applications. Modern, state-of-the-art, efficient Datalog engines have been built, such as Soufflé and LogicBlox. These modern engines provide very high performance on a single node using thread-safe data structures such as a concurrent B-tree [?] and other carefully engineered data structures. However, they do not support the distribution of computation over multiple machines.

A Datalog program can be represented as a collection of rules, and executing a program amounts to the evaluation of these rules. The process of evaluating rules is embarrassingly parallel [2] and can be done independently on multiple machines. This can be achieved, for example, with the MapReduce [3] programming model where the evaluation of each rule is the map task and aggregation into a knowledgebase is the reduce task.

This project will be implemented in Erlang. Erlang is a functional language that is commonly used to build massively scalable soft real-time systems with high availability requirements. It has good support for concurrency and error handling, which makes it a good fit for this project.

APPENDIX D. PROJECT PROPOSAL

```

1 r1 ancestor(X,Y) :- parent(X,Y).
2 r2 ancestor(X,Y) :- parent(X,Z), ancestor(Z, Y).

```

Listing 1: Minimal example of a recursive query program.

1.1 Datalog evaluation background

1.1.1 Serial Datalog engine

Traditionally Datalog is implemented by the bottom-up evaluation method, which is in contrast to the top-down method used by Prolog. It repeatedly evaluates rules until a fixpoint is reached. The simplest way to compute the fixpoint is the naïve evaluation strategy. This strategy repeatedly applies rules to the database instance, initially the source input database, called the extensional database predicates (EDB), to derive new tuples. This process is repeated until no new tuples can be derived, in other words, a fixpoint is reached. Table 1 shows the example of computing the fixpoint of the program given in Listing 1.

The semi-naïve evaluation strategy builds on top of the naïve strategy by keeping track of a set of relations *deltas*, which are the new tuples generated during each iteration and only evaluate rules on this set when generating new tuples.

parent		ancestor	
X	Y	X	Y
a	b	a	b
b	c	b	c
c	d	c	d
(a) initial input parent relation		(b) iteration 1	

ancestor		ancestor	
X	Y	X	Y
a	b	a	b
b	c	b	c
c	d	c	d
a	c	a	c
b	d	b	d
a	d	a	d
(c) iteration 2		(d) iteration 3	

Table 1: An example of computing the fixpoint of a query, fixpoint will be reached in iteration 4, shaded rows are new tuples derived.

To clarify the terminologies for Datalog before we talk about the representation, a term in Datalog is either a constant or a variable, and an atom (or goal) is a predicate symbol (function) along with a list of terms as arguments [6]. This use of terminologies is different from Prolog.

This process of building a serial engine can be implemented via relational algebra (RA). We can represent each atom as a relation and evaluation becomes the database operations such as projection, selection, renaming and join. For example, in program 1 given above, we can see that the rule r2 can be computed by a join operation $\text{parent} \bowtie_{2=1} \text{ancestor}$ on the second and first

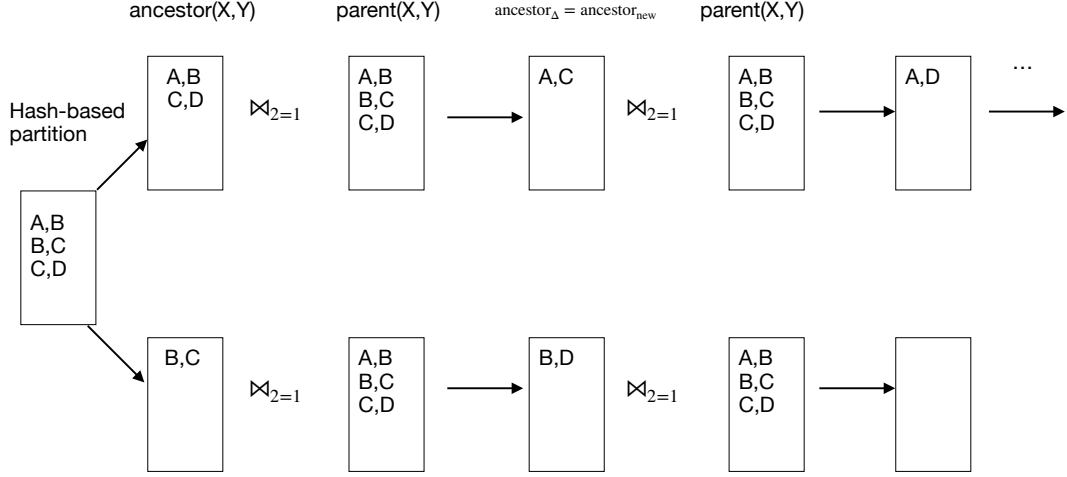


Figure 1: Example of distributing relations across workers, the final aggregation is not shown in the graph.

column of parent and ancestor.

1.1.2 Distributing the work

Because the underlying representation of the primitive operations in RA is inherently parallel, we can extract data-parallelism from this process by decomposing the RA operations over multiple workers. There are several ways of doing this, perhaps the most straightforward one, which is also the main one to be used in this project, is to use the MapReduce programming model. This means that there will be a master responsible for distributing relations to be evaluated to workers and each worker would do its evaluation independently. The result produced by workers is then aggregated into a single database instance.

Figure 1 shows an example of distributing relations to two workers. Continuing the example of the ancestry program above, we can partition the initial relation into several sub-relations (for example, using a hash-based partition), and distribute them over to multiple processes. Each process then performs join independently and finally, there will be a final union operation (reduce) that collects all the results together.

2 Subtasks

This project can be divided into the following subtasks:

Research and preparation

1. Learn ways of implementing a serial Datalog evaluator with primitive Database operations.
2. Define a set of operations that the core Datalog language is going to support.
3. Learn how to distribute the work using hash-based joins and hash-based distribution of relations to multiple workers.

Implementation

1. Build a parser that can parse the core of Datalog syntax, in particular, rules of the form:

$A :- B_1, B_2, \dots, B_n.$

APPENDIX D. PROJECT PROPOSAL

Where A and B are Datalog atoms defined in Subsection 1.1.1. This can be done using a lexer and parser generator. In this project, I will be using `leex` and `yacc` which are the lexer and parser generator that comes with Erlang.

2. Implement a serial Datalog engine with the naïve evaluation strategy first and then improve it with the semi-naïve strategy, as mentioned in Subsection 1.1.1.
3. Extending the serial engine so that it can distribute work among multiple workers using the method described in Subsection 1.1.2.

Evaluation

1. Generate synthetic datasets that can represent the typical workload of a Datalog program. The Soufflé repository has some benchmarking data generating script [8], which can be borrowed to evaluate this project as well.
2. Use profiling tools on the engine that is run with both the synthetic and real input data to measure the throughput of the engine. Erlang provides a series of profiling tools such as `fprof` and `eprof` and benchmarking facilities such as `timer:tc` that can measure clock time.
3. Writing code that can simulate worker failures and run benchmarking on it with worker failures.

Writing up

1. Writing up the progress report and prepare for the presentation.
2. Writing up the dissertation.

3 Starting point

This project will be implementing the Datalog engine from scratch in Erlang. I have no prior experience working with Erlang apart from reading the first few chapters of *Learn You Some Erlang for Great Good!* [9] and did some exercise in the book during the summer holiday.

I do not have any prior experience writing an interpreter/compiler for a (logic) programming language either, apart from the IB Compiler Construction and the Prolog course.

I do have experience writing a toy MapReduce application that counts the number of words in several documents for personal interest. There are similar concepts that can be used in this project, but I will not use any code from this application.

3.1 Other related work

There are already well-built Datalog engines such as Soufflé, which has very high performance on a single node, but does not offer distributed computation. This project might borrow ideas from Soufflé implementation techniques but will not use any of its codebase.

There have already been attempts to build data-parallel Datalog, such as Google's Yedalog [?] which has a batch backend that compiles the program into Flume pipelines [1]. And also attempts that utilises the parallel relational algebra [5]. This project takes an approach more similar to Yedalog and leaves the parallel relational algebra approach as an extension.

4 Success Criterion

This project will be considered successful if a distributed Datalog engine is produced that is capable of distributing its work across multiple workers. It should be able to compute the transitive closure, that is, the smallest relation that contains R and is transitive, of a reasonably sized graph (containing at least 100 nodes). Listing 1 is an example program that computes the transitive closure.

And this core deliverable will be evaluated in two ways:

scalability Answering the question: how does adding more workers change the throughput (the amount of data processed in a given time) of the engine? This will be done by adding more workers to the engine and a graph of throughput against the number of workers can be plotted.

fault tolerance Answering the question: how does the engine tolerate worker failure? this can be evaluated by killing workers randomly, and check the correctness of the output against the serial engine. A graph of throughput against the number of worker failures will also be plotted.

In terms of the input data for evaluation, I will first generate synthetic datasets that represent different workloads (those that can be easily parallelised and those that cannot) and see how the engine performs under different loads.

Then I will use the Arxiv High-Energy Physics paper citation network (or a scaled-down version of it) [4] to test the performance of the engine on a real network by computing the transitive closure on this graph.

5 Possible extensions

This section contains some possible extensions that are not part of the success criteria, but will be implemented and evaluated if there is additional time.

- Running this engine on real distributed machines and measure its performance against the number of physical machines. This can be compared with evaluation using processes and see how network delay and communication overhead can impact the performance of the engine.
- The core Datalog semantics does not include negation. One popular way of doing that is through *stratified negation*. This requires constructing a precedence graph of the program and determine whether this program is *stratifiable*. This precedence graph can be used to determine the order in which relations are computed and this can be used to impose an order in which tasks get assigned to each worker, and its impact on scalability can be measured.
- In the original design of the MapReduce model, backup tasks are used to handle the problem of a “straggler”: a particularly slow worker. It works by scheduling backup executions of the remaining *in-progress* tasks when the whole task is close to completion. [3]. This can be implemented as an extension to the distributed engine as well and evaluate its impact on the overall performance.
- Simple hash based distribution does not guarantee that each partition will be of the same size, this might lead to problems such as assigning one worker too much work to do. *Balanced parallel relation algebra* (BPRA) [5] solves this problem by having a two-layered distributed hash-table. This extension is therefore to implement the distribution of relations using

APPENDIX D. PROJECT PROPOSAL

BPRA and compare its scalability with the hash-based join and distribution implemented in the core part.

6 Work plan

18th Oct, 2021 – 31st Oct, 2021

Read relevant papers on how to implement a serial and a parallel Datalog engine, making design choices on representations of Datalog rules and facts.

Read relevant sections of *Learn you some Erlang for great good* and get more familiar with the concurrency aspect of the language.

Milestone: Sent supervisor a report detailing the design of a serial/parallel Datalog engine, including design choices and ask for feedback.

1st Nov, 2021 – 14th Nov, 2021

Write a minimal parser that can parse Datalog rules into a representation that can be used by the evaluator. As this is not the main focus of the project, if it takes too much time, then skip it and focus on writing the evaluator.

Implement an engine that uses the naïve evaluation strategy.

Milestone: Produced a minimal working Datalog engine that can pass the litmus test of logic programming¹ with support for the core part of Datalog. But no support for distribution, and no need to consider performance at this stage.

15th Nov, 2021 – 28th Nov, 2021

Implement the semi-naïve evaluation strategy.

Milestone: Have a working implementation of a Datalog engine with the semi-naïve evaluation method and can pass the litmus test.

29th Nov, 2021 – 12th Dec, 2021

1st Dec, 2021 *Last day of lecture*

Write more extensive tests for the current engine, covering edge cases, invalid inputs, etc.

Fix bugs uncovered by the tests.

Milestone: Having a more extensive test framework that can cover the basic functionalities of the code written so far.

This work package is intentionally made lightweight to accommodate two deadlines of the unit of assessment in these two weeks.

13th Dec, 2021 – 26th Dec, 2021

Add distribution support for the existing Datalog engine and verify its functionality using the same test for the serial engine.

Milestone: Have a basic working distributed Datalog engine that can produce the same result as the sequential Datalog engine on simple inputs.

¹compute a variant of the ancestry program, i.e. the transitive closure of a relation

27th Dec, 2021 – 9th Jan, 2022

Slack/buffer period

Milestone: A basic working distributed engine should have been produced that can pass the tests written so far.

10th Jan, 2022 – 23rd Jan, 2022

Plan evaluation.

Perform any necessary setups for evaluation/writing infrastructure for evaluation.

Evaluation of the core distributed engine.

Milestone: Written down a plan of the evaluation and sent this to the supervisor.

Milestone: Have some basic evaluation results of the core part, including at least the scalability of the engine.

24th Jan, 2022 – 6th Feb, 2022

Preparing for the Progress report and presentation.

Milestone: Deliver a presentation of the progress made so far.

4th Feb, 2022 *Progress report deadline*

7th Feb, 2022 – 20th Feb, 2022

10th Feb, 2022 *Progress Report Presentations*

Start writing the introduction and preparation chapter of the dissertation. Sending them to the supervisor for feedback.

Finish evaluating of the core part of the engine.

Milestone: Draft introduction and preparation complete and sent to the supervisor and/or DoS.

Milestone: Evaluation of the core part of the engine finished.

21st Feb, 2022 – 6th Mar, 2022

Slack + buffer period.

Milestone: Evaluation of the core part of the project should be completed, data collected, graph plotted, etc.

7th Mar, 2022 – 20th Mar, 2022

Start writing the implementation chapter of the dissertation.

Start working on the first/second extension.

Milestone: Draft implementation chapter complete and sent to the supervisor and/or DoS.

APPENDIX D. PROJECT PROPOSAL

21st Mar, 2022 – 3rd Apr, 2022

Start writing the evaluation and conclusion chapter of the dissertation.

Start evaluating the extension work done so far.

Milestone: Draft dissertation complete and sent for feedback.

Milestone: Collected basic results of the extension evaluation.

4th Apr, 2022 – 17th Apr, 2022

Make improvements based on feedback from the supervisor and finish off the dissertation.

Adding new content based on the result of the extension and its evaluation.

Milestone: Submitted the first draft of the dissertation.

18th Apr, 2022 – 1st May, 2022

Final changes to the dissertation.

Milestone: Dissertation submitted.

2nd May, 2022 – 15th May, 2022

Slack period.

Milestone: Dissertation submitted.

17th May, 2022 *Dissertation deadline*

7 Resource declaration

I will be primarily using personal laptop for development:

- Model: MacBook Pro 2020
- Processor: 2 GHz Quad-Core Intel Core i5
- Memory: 16 GB 3733 MHz LPDDR4X

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. I will be backing up my entire disk bi-weekly using Apple's time machine to an external hard drive. My repositories and dissertation will also be pushed up to Github after each commit. And I will also be using Google Drive to periodically backup my project directory. In case of hardware failure, I will be using MCS and/or the remote shell service provided by the Studnet-Run Computing Facility (SRCF). Erlang is installed on the shell server provided by SRCF and it also has the text editor (`vim`) I need.

I have downloaded the evaluation dataset mentioned in Section 4 from [7] onto my local machine and will be backed up along with the rest of the project directory.

In terms of evaluation, I will also be using my own machine as the main platform to evaluate the engine. Evaluating on a real large multi-core machine will be left as an extension if there is enough time and budget. For evaluating the project on real distributed machines (which is also in the extension) I plan to use AWS and use student credit to access their service.

References

- [1] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan. Flume-java: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010. URL <http://dl.acm.org/citation.cfm?id=1806638>.
- [2] M. Contrastin. The essence of datalog, June 2021. URL <https://dodisturb.me/posts/2018-12-25-The-Essence-of-Datalog.html>.
- [3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.
- [4] J. Gehrke, P. Ginsparg, and J. Kleinberg. Overview of the 2003 kdd cup. *SIGKDD Explor. Newsl.*, 5(2):149–151, Dec. 2003. ISSN 1931-0145. doi: 10.1145/980972.980992. URL <https://doi.org/10.1145/980972.980992>.
- [5] T. Gilray, S. Kumar, and K. Micinski. Compiling data-parallel datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 23–35, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383257. doi: 10.1145/3446804.3446855. URL <https://doi.org/10.1145/3446804.3446855>.
- [6] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. *Datalog and Recursive Query Processing*. Now Foundations and Trends, 2013.
- [7] KDD Cup. Citation graph of the hep-th portion of the arxiv, 2003.
- [8] Soufflé developers. Soufflé benchmarks, 2021. URL <https://github.com/souffle-lang/benchmarks.git>.
- [9] F. Trottier-Hebert. *Learn You Some Erlang for Great Good!* online, 2013. URL <https://learnyousomeerlang.com>.