Declarative Systems

Tyson Condie

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-71
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-71.html

June 4, 2011

Declarative Systems

by

Tyson Condie

A dissertation submitted in partial satisfaction

of the requirements for the degree of

Doctor of Philosophy
in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph M. Hellerstein, Chair
Professor Michael J. Franklin
Professor Tapan S. Parikh
Professor Raghu Ramakrishnan
Professor Ion Stoica

Spring 2011

Declarative Systems

Abstract

Declarative Systems

by

Tyson Condie

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

Building system software is a notoriously complex and arduous endeavor. Developing tools and methodologies for practical system software engineering has long been an active area of research. This thesis explores system software development through the lens of a declarative, data-centric programming language that can succinctly express high-level system specifications and be directly compiled to executable code. By unifying specification and implementation, our approach avoids the common problem of implementations diverging from specifications over time. In addition, we show that using a declarative language often results in drastic reductions in code size (100× and more) relative to procedural languages like Java and C++. We demonstrate these advantages by implementing a host of functionalities at various levels of the system hierarchy, including network protocols, query optimizers, and scheduling policies. In addition to providing a compact and optimized implementation, we demonstrate that our declarative implementations often map very naturally to traditional specifications: in many cases they are line-by-line translations of published pseudocode.

We started this work with the hypothesis that declarative languages — originally developed for the purposes of data management and querying — could be fruitfully adapted to the specification and implementation of core system infrastructure. A similar argument had been made for networking protocols a few years earlier [61]. However, our goals were quite different: we wanted to explore a broader range of algorithms and functionalities (dynamic programming, scheduling, program rewriting, and system auditing) that were part of complex, real-world software systems. We identified two existing system components — *query optimizers* in a DBMS and *task schedulers* in a cloud computing system — that we felt would be better specified via a declarative language. Given our interest in delivering real-world software, a key challenge was identifying the right system boundary that would permit meaningful declarative implementations to coexist within existing imperative system architectures. We found that *relations* were a natural boundary for maintaining the ongoing system state on which the imperative and declarative code was based, and provided an elegant way to model system architectures.

This thesis explores the boundaries of declarative systems via two projects. We begin with Evita Raced; an extensible compiler for the Overlog language used in our declarative networking system, P2. Evita Raced is a metacompiler — an Overlog compiler written in Overlog — that integrates seamlessly with the P2 dataflow architecture. We first describe the minimalist design of Evita Raced, including its extensibility interfaces and its reuse of the P2 data model and runtime engine. We then demonstrate that a declarative language like Overlog is well-suited to expressing traditional and novel query optimizations as well as other program manipulations, in a compact and natural fashion. Following Evita Raced, we describe the initial work in BOOM Analytics, which began as a large-scale experiment at building "cloud" software in a declarative language. Specifically, we used the Overlog language to implement a "Big Data" analytics stack that is API-compatible with the Hadoop MapReduce architecture and provides comparable performance. We extended our declarative version of Hadoop with complex distributed features that remain absent in the stock Hadoop Java implementation, including alternative scheduling policies, online aggregation, continuous queries, and unique monitoring and debugging facilities. We present quantitative and anecdotal results from our experience, providing concrete evidence that both data-centric design and declarative languages can substantially simplify systems programming.

To Joe, Paul, and Lara.

Contents

Acknowledgements

I would like to thank my advisor Professor Joe Hellerstein for helping me achieve this goal. I was first introduced to Joe by Mike Franklin, who recommend me for a TA position in the database systems course (CS186) at U.C. Berkeley. Joe exemplified the role of a Professor and inspired me to enter graduate school. As a graduate student, Joe went above and beyond the call of an advisor; working with me on developing my research, writing, and presentation skills. I could not have asked for a better role model. I would also like to thank Professors Mike Franklin, Ion Stoica, and Scott Shenker for their guidance during my graduate career.

I spent my first two years of graduate school at the Intel Research Berkeley Lab. I would like to thank everyone in this lab for their assistance during the early stages of my graduate career. Two research scientists immediately stand out: Petros Maniatis and Timothy Roscoe. I will always consider Petros my second advisor and I thank him for spending countless hours helping me develop my research and building my confidence. Mothy introduced me to systems research and he is the best, and no doubt wittiest, systems programmer I have had the privilege of working with.

I am thankful for my two summer internships at Yahoo! Research; a place I now call home. My mentor Christopher Olston was a true inspiration in my life. I strive to emulate Chris' approach to research and I thank him for his guidance. I would also like to thank Benjamin Reed, Khaled Elmeleegy, Vanja Josifovski, Raghu Ramakrishnan, Utkarsh Srivastava, and Adam Silberstein for their influence on my research.

I had the privilege of working with some of the best graduate students that computer science has to offer. I would like to thank Boon Thau Loo for mentoring me in the P2 project and showing me the path to a successful graduate career. My time at Intel Research Berkeley would not have been successful without the company of my dear friend Atul Singh, who was always there for me in times of need. In the latter stages of my graduate career, I worked with Peter Alvaro and Neil Conway (the dream team) on BOOM Analytics and MapReduce Online. I would like to specifically thank Neil Conway for his co-first authorship on the MapReduce Online paper, and Peter Alvaro for his guidance in BOOM Analytics and for always saying something is good before giving me constructive criticism. Russell Sears also played a key role in the development of these two projects and provided an irreplaceable presence to the success of the BOOM team. I would like to thank Kuang Chen for showing what research with real-world impact is all about and for his friendship. Thanks to Matei Zaharia for instructing me on Hadoop. To Alexandra Meliou for helping me pass my preliminary examination and her leadership role in the database group. To David Chu, Shawn Jeffery and Ryan Huebsch for helping me develop my research topic. And the many others that had an impact on my life at Berkeley: Daisy Wang, Sean Rhea, Matthew Caesar, Beth Trushkowsky, Michael Armbrust, Fred Reiss, Mehul Shah, Ashima Atul, and William Marczak.

Two experiences prior to entering graduate school at U.C. Berkeley continued to shape and motivate me. First, I had the privilege of working with Professor Hector

Garcia-Molina at Stanford University. Hector introduced me to computer science research and worked with me on my first publication. Sep Kamvar, Mayank Bawa, and Prasanna Ganesan also played a pivotal role in this early stage of development. Second, I am grateful for my experience in the U.S. Marine Corps, which motivated me toward a higher degree — Semper Fidelis.

Finally, I would like to dedicate this thesis to the three individuals that made it possible. To Professor Joe Hellerstein for navigating me through this incredible experience and teaching me how to stand on my own two feet. To my uncle Paul Condie for inspiring me to enter this field and teaching me how to think outside of the box. To my wife Professor Lara Dolecek for leading me by example and supporting me from the very beginning.

Chapter 1

Dissertation Overview

There has been renewed interest in recent years on applying declarative languages to a variety of applications outside the traditional boundaries of data management. Examples include work on compilers [56], computer games [96], security protocols [57], and modular robotics [11]. Our work in this area began with the *Declarative Networking* project, as instantiated by the *P2* system for Internet overlays [63, 62]. The P2 project demonstrated the viability of declarative languages as being a natural fit for programming network overlay protocols. In Chapter 2, we review this influential work because it sets the stage for this thesis. Specifically, we describe the declarative language Overlog — a dialect of Datalog — and the P2 system, which compiles Overlog programs into dataflow runtime implementations reminiscent of traditional database query plans.

Following the background material, Chapter 3 describes a declarative system component called Evita Raced, which is a metacompiler implemented in P2. Evita Raced formulates the task of query compilation as a query; written in the same declarative language (Overlog) used by "client" queries: such as the various networking protocols from Loo, et al. [62, 63]. Evita Raced exposes the P2 compiler state to the Overlog language (Chapter 3.1), thereby permitting the specification of query transformations (i.e., optimizations) in Overlog. Many traditional database optimizations, like the magic-sets rewrite (Chapter 4), the System R dynamic program (Chapter 5.2), and the Cascades branch-and-bound algorithm (Chapter 5.3), can be fully expressed as Overlog queries. Specifying these optimizations as Overlog queries results in a more concise representation of the *algorithm as code* and a dramatic reduction in the overall development effort. We reflect on the practicalities of a declarative approach to query compilation and our overall experience with Evita Raced in Chapter 6.

In Chapter 7, we turn to the topic of *cloud computing* [10] and introduce the BOOM project: an effort to explore implementing cloud software using declarative, data-centric languages. As a first concrete exercise, we built BOOM Analytics: an API-compliant reimplementation of Hadoop MapReduce in a declarative language. In Chapter 8, we review the salient aspects of Hadoop and the MapReduce programming model that it implements. In Chapter 9, we describe our rewrite of the Hadoop sched-

uler in a declarative language and show that equivalent performance, fault-tolerance, and scalability properties can be achieved in a declarative language. In Chapter 10, we evolve the batch-oriented data flow implemented by Hadoop to a more online execution model that pipelines data between operators. We then describe extra scheduling policies implemented in the declarative scheduler that accommodate pipelined plans. Finally, we conclude in Chapter 11 with a discussion of future directions.

Chapter 2

P2: A Logical Beginning

This chapter contains background material related to the *Declarative Networking* project [61], which is a lead-in to this thesis. The original project members included Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, and Ion Stoica at the *University of California, Berkeley*, Petros Maniatis and Timothy Roscoe at *Intel Research Berkeley*, and Raghu Ramakrishnan at *Yahoo! Research*. Together, we developed a new declarative language called *Overlog* and a runtime system called *P2*. Our initial goal was to make it easy to implement and deploy *overlay networks* [1] by allowing specifications in a high-level declarative language to be directly executed on nodes that span the Internet. These overlay specifications, expressed as Overlog rules, contained *orders of magnitude* fewer lines of code than the corresponding overlay implementations written in an imperative language (e.g., C/C++). The project implemented, and deployed, declarative versions of a Narada-style mesh network [25], using only 12 "rules", and the Chord structured overlay [89] in only 35 "rules" [63]. The P2 project clearly showed that relations, together with a recursive query language, can fairly naturally represent the persistent routing state of the overlays it considered [61].

The Overlog language is a descendent of Datalog, which we review in Chapter 2.1. In Chapter 2.2, we present the Overlog language by detailing its extensions to Datalog: it adds a notation to specify the location of data, provides some SQL-style extensions such as primary keys and aggregation, and adds a flexible notion of state lifetime. Chapter 2.3 describes the P2 runtime, which is responsible for compiling and executing Overlog programs on a set of distributed nodes. The design of P2 was inspired by prior work in programming languages [85], databases [66, 92, 21, 46], systems [79], and networking [89, 54]. The P2 implementation is based in large part upon a side-by-side comparison of the PIER peer-to-peer query engine [46] and the Click modular router [54]. Like PIER, P2 can manage structured data tuples flowing through a broad range of query processing elements, which may accumulate significant state and perform substantial asynchronous processing. Like Click, P2 stresses high-performance transfers of data units, as well as dataflow elements with both "push" and "pull" modalities.

[1] A computer network built on top of an existing network e.g., IP (layer 3).

```
link(``node1'', ``node2'', 1).
link(``node2'', ``node3'', 1).

r1 path(X, Y, cons(X, Y), C) :-
   link(X, Y, C).

r2 path(X, Z, cons(X, P2), C1+C2) :-
   link(X, Y, C1), path(Y, Z, P2, C2),
   contains(X, P2) == false.

query path(``node1'', Y, P, C).
```

Figure 2.1: Path program written in Datalog.

2.1 Introduction to Datalog

Our description of Datalog is based on a survey by Ramakrishnan and Ullman [76], and course notes [93] on the subject. Datalog drew inspiration from the Prolog language, which was one of the first logic programming languages. Both Datalog and Prolog consist of a set of declarative *rules* and an optional *query*. A rule has the form p `:-` q_1, q_2, ..., q_n, which informally reads "**if** q_1 and q_2 and ... and q_n is true **then** p is true." The predicate appearing to the left of the `:-` symbol is the head predicate, and those to the right are body literals or "subgoals." Literals are either *predicates* over *fields* (variables and constants), or function symbols applied to fields. Recursion is expressed by rules that refer to each other in a cyclic fashion. That is, the head predicate also appears as a subgoal in the rule, or indirectly through some other subgoal predicates.

A predicate literal is a named reference to a set of data tuples associated with a specific schema. In Datalog, a data tuple is referred to as a *fact*, which is stored in a relational table that may not necessarily fit in memory. A predicate whose relation is stored in the database is called an *extensional database* (EDB) relation, while those that are defined by logical rules are called *intensional database* (IDB) relations. In other words, EDB tuples are those that persist in the database as relations, while IDB predicates are more like "views" (or stored queries) over the database schema.

During evaluation, EDB facts represent the input to the Datalog program, and IDB derivations are the output. Most implementations of Datalog evaluate rules in a bottom up fashion, starting with all known EDB facts, and deriving new IDB facts through rule deductions. A key consequence of a bottom-up evaluation strategy is that it can efficiently handle relations whose size exceed the capacity of a machine's main memory.

2.1.1 Datalog Syntax

Figure 2.1 provides our first look at a program expressed in Datalog. The *fact* statements at the top specify the existence of two data tuples in the `link` table with the given attribute constants. Each row of the `link` table contains three attributes; two strings, and an integer. The program rules derive all reachable paths from this initial set of known `link` tuples, and presents that result in a relational view called `path`.

Base derivations proceed from the rule body (those predicates to the right of "`:-`") and project onto the rule head (to the left of "`:-`"). The `link` facts are used in the evaluation of rule `r1` to derive an initial set of `path` tuples. The rule reads "if there exists a `link` from X to Y at cost C, then there exists a `path` from X to Y consisting of nodes X, Y at cost C." Both initial facts meet this criterion and hence are included in the `path` relation.

Rule `r2` expresses a transitive closure over the `link` and `path` relations. The rule reads "if there is a link from X to Y at cost C, and there is a `path` $P2$ from Y to Z at cost $C2$, then there is a path from X to Z via $P2$ at cost $C1 + C3$." A path from "node1" to "node3", through "node2", satisfies this criterion, and such a tuple is included in the `path` IDB relation. The selection predicate `contains(X, P2) == false` avoids cyclic paths, ensuring a finite result and program termination. The "query" predicate at the bottom of Figure 2.1 asks for all paths that start at "node1." The `path` tuples that begin with "node1" and end at "node2" and "node3" (via "node2") both meet this query constraint.

2.1.2 Safety First

There are constraints that must be in place for a Datalog program to make sense as operations on finite relations.

Definition 1. *A* safe *Datalog rule ensures that all variables mentioned in the rule appear in some nonnegated subgoal table predicate of the rule body.*

This definition ensures that all variables in negated subgoals and the head predicate are restricted by some nonnegated subgoal table predicate. For example, the following rule is not safe since it does not restrict the P variable in the `path` head predicate.

```
path(X, Y, P, C) :- link(X, Y, C).
```

The above rule generates an infinite number of `path` tuples since we can substitute any conceivable value for P. A safe Datalog rule is a necessary, but not sufficient, [2] condition for obtaining a finite (IDB) solution from evaluating a finite set of rules on a finite (EDB) input. Datalog further restricts its (IDB) output to set semantics, as

[2]The programmer can still express an infinite solution, for example by simply leaving out the `contains(X, P2) == false` predicate in rule `r2`

1: $\texttt{path} = \Delta\texttt{path} = \pi_{X,Y,cons(X,Y),C}\texttt{link}$
2: **while** $\Delta\texttt{path}$!= $\emptyset$ **do**
3: $\quad\Delta\texttt{path} = \pi_{X,\ Z,\ cons(X,P2),\ C1+C2}\ (\sigma_{contains(X,P2)\ ==\ false}(\texttt{link} \bowtie \Delta\texttt{path}))$
4: $\quad\Delta\texttt{path} = \Delta\texttt{path}$ - path
5: $\quad\texttt{path} = \texttt{path} \bigcup \Delta\texttt{path}$
6: **end while**

Figure 2.2: Seminaïve evaluation of the path program in Figure 2.1.

opposed to bag semantics that allow duplicate tuples. The reader can assume these safety restrictions in all the rules presented in this thesis.

2.1.3 Evaluation of Datalog Rules

We now turn to the evaluation of a set of Datalog rules, which is performed in a bottom-up fashion, starting with the set of known EDB facts. There are two standard approaches to evaluating a set of Datalog rules. The first is called *naïve evaluation*, which is an iterative algorithm that repeatedly applies all known facts to the program rules, in some stylized set-oriented fashion, until no new knowledge is obtained. Starting with the tuples contained in the EDB, the naïve evaluator iteratively executes a select-project-join (SPJ) query against the predicates in the rule body, to continually derive new IDB tuples. Each iteration applies all the tuples contained in the EDB and IDB to the rule set. The process repeats until no new tuples can be inferred, marking the end of the evaluation, which is commonly referred to as a "fixed point."

Each iteration of this naïve algorithm uses all known data in the database when deriving new data. A second approach, which is also the optimal approach, adds a condition to the iteration loop that prunes the data that was not derived in the previous round. The remaining facts, if any, are then used in the subsequent iteration. This *seminaïve evaluation* algorithm is based on the principle that "if a fact is derived during round i then it must have been inferred from a rule in which one or more subgoals were instantiated with facts that were inferred in round $i-1$." [94]

Figure 2.2 describes the steps performed by a seminaïve evaluation of the path program shown in Figure 2.1. Let Δ`path` be a reference to the set of new tuples added to the `path` relation in the previous round. In the first round, line 1 of the algorithm uses rule `r1` to derive the initial set of `path` tuples from the EDB `link` relation. Subsequent rounds are carried out in the `while` loop (lines $2 \cdots 6$) until Δ`path` is empty. The body of the loop contains the following three steps.

1. Evaluate rule `r2` relative to the tuples in Δ`path`.
2. Assign Δ`path` to the new tuples derived in this round only.
3. Accumulate the new Δ`path` in the `path` relation.

```
r4 path(X, Y, cons(X, Y), C) :-
   link(X, Y, C),
   not detour(X, Y).

r5 detour(X, Y) :-
   link(X, Y, C),
   ...
```

Figure 2.3: Negated Datalog rule.

In the first step, we evaluate rule `r2` using the Δ`path` tuples derived in the previous round (e.g., initially, those obtained from the `link` relation). In general, if there existed other rules that referenced `path` in the body, then those too would be evaluated against the same Δ`path` tuples, and any deductions would contribute to the Δ predicate referenced by the rule head. The last two steps in the loop deal with ensuring Δ`path` only references deductions from the previous round, and that the new deductions are accumulated in the `path` relation.

2.1.4 Fixed Point Semantics

Datalog is a monotonic language: once a fact is derived during evaluation it is certain to be in the final answer. The evaluation of a program proceeds as a series of deductions to the IDB. A Datalog program is said to be at a *fixed point* when no further deductions can be made relative to the current EDB and IDB tuples. The result derived at a fixed point is a model for the Datalog program. Given a model m and a Datalog program p, m is a minimal model if and only if no proper subset of m is a model for p. In the absence of negated subgoals, a Datalog program has one and only one minimal model. A logic program with negation may have more than one minimal model. However, if the program is "stratified" then there is a uniquely identifiable "intended" minimal model, based on the (stratification) order in which the relations are (intended to be) minimized. [76]

2.1.5 Negation and Stratification

We touch on the subject of handling negated subgoals in the body of a Datalog rule. There is a large body of work on this subject that we will not address here since it does not pertain to the content of this thesis. Our goal instead is to introduce the reader to the notion of stratified negation, which ensures that a set of Datalog rules with negated subgoals "make sense," by way of reaching an intended minimal model on a fixed point evaluation. Before going further, we review some semantic issues raised by negated subgoals in Datalog.

Consider rule `r4` in Figure 2.3, which formulates a `path` from a `link` if X and Y does not cross a detour. Unfortunately, the complement of the `detour` relation is not

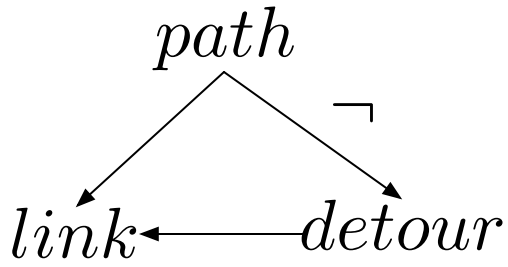


Figure 2.4: Dependency graph for predicates appearing in Figure 2.3.

well-defined; since the variables range over an infinite domain the compliment is also infinite. Moreover, we cannot specify the complete `detour` relation prior to evaluation, since it is an IDB predicate (due to rule `r5`). If we were to simply evaluate the rules in Figure 2.3 (using for example the seminaïve algorithm) then we could end up with `path` tuples that cross detours. To see this, lets assume that we start by evaluating rule `r4` with the initial facts in the `link` relation. The execution plan for the negated `detour` is similar to an anti-join operation, where tuples from `link` relation pass if they do not already exist in the current `detour` relation. Since we have not yet evaluated rule `r5`, all `link` tuples pass the anti-join, and produce a set of `path` deductions in rule `r4`. Subsequently evaluating rule `r5` would give us our `detour` tuples, but this would be too late in the sense that we have already made incorrect deductions, and cannot take them back. [3]

We could obtain the correct IDB by simply evaluating rule `r5` first. Such an ordering of predicate evaluations forms the basic idea behind stratified Datalog. Before we get to that definition, we first review how the dependencies of a Datalog program are represented graphically. Figure 2.4 shows the dependency graph for the predicates appearing in the rules of Figure 2.3. Constructing this graph is a straightforward application of the following two rules.

1. Add $p \rightarrow q$ dependency if there is a rule with head predicate p and subgoal q.
2. Add $p \rightarrow q$ dependency labeled $\neg$ if there is a rule with head predicate p and negated subgoal q.

From Figure 2.3, rule `r4` forms the $path \rightarrow link$ and negated: $path \rightarrow detour$ edge dependencies, while rule `r5` supplies the $detour \rightarrow link$ edge dependency.

The stratum of an IDB predicate p is defined to be the largest number of negations ($\neg$) along any path involving predicate p. The dependency graph in Figure 2.4 places predicates `detour` and `link` in the lowest stratum 0, while the `path` predicate is in stratum 1. If all IDB predicates have a finite stratum, then the Datalog program is *stratified*. If any IDB predicate has an ∞ stratum, then the program is unstratified. An IDB predicate is assigned an ∞ stratum if it is included in a cyclic path crosses a negated (subgoal) edge.

[3] Recall: Datalog is a monotonic language (Chapter 2.1.4).

```
link(``node1'', ``node2'', 1).
link(``node2'', ``node3'', 1).

r1 path(X, Y, cons(X, Y), C) :-
   link(X, Y, C).

r2 path(X, Z, cons(X, P2), C1+C2) :-
   link(X, Y, C1), shortestPath(Y, Z, P2, C2),
   contains(X, P2) == false.

r3 minCostPath(X, Y, min<C>) :-
   path(X, Y, P, C).

r4 shortestPath(X, Y, P, C) :-
   minCostPath(X, Y, C), path(X, Y, P, C).
```

Figure 2.5: Shortest path variant of Figure 2.1.

We evaluate a stratified Datalog program using the seminaïve algorithm (e.g., Figure 2.2) but with a slight twist – we sort the IDB predicates by their assigned stratum, and follow this order when choosing Δ predicates (e.g., Δ `path`) to evaluate in the loop. This order ensures that if the program is stratified then any negated subgoal (i.e., `detour`) has already had its relation fully evaluated first. The result of this evaluation is called a *stratified model*. [4]

We revisit the notion of stratified Datalog throughout this thesis. It turns out that the P2 system did not supported stratified Datalog, which slightly complicated the (Overlog) program rules described in Chapters 3, 4 and 5. Fortunately, there is another class of *locally stratified* Datalog programs that "make sense" on certain data.

2.1.6 Local Stratification

Stratified Datalog is defined in terms of a syntactic property that translates to cycles through negations in the dependency graph of a collection of rules. An extension to this definition is a class of locally stratified programs, which is defined in terms of a data dependent property. Intuitively, these programs are not necessarily stratified according to their rules, but they are stratified when we instantiate those rules on a specific collection of data. Many of the rules and data instances presented in this thesis fall into the class of locally stratified programs.

Like negation, an aggregation adds a stratification boundary to a Datalog program. Intuitively, we must derive all facts from the tables mentioned in the rule body, before we can evaluate the aggregate in the rule head. Consider the variant of the path program in Figure 2.5, which modifies rule `r2` to formulate new paths from the

[4] We further note that the notion of stratified Datalog has nothing to do with the termination of a Datalog program. The issue here is the existence a unique minimal result that is consistent with the programmer's intent.

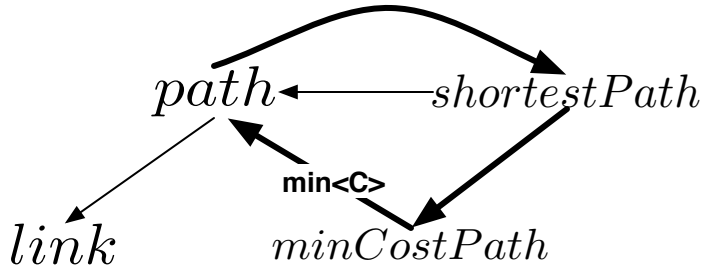


Figure 2.6: Dependency graph for predicates appearing in Figure 2.5. A cycle through an aggreagation appears in bold.

`shortestPath` relation, rather than the `path` relation. Two extra rules `r3` and `r4` are used to derive the shortest path from the `path` relation. Rule `r3` selects the minimum cost path from X to Y, and rule `r4` selects the actual minimum path based on the minimum cost value in C.

The dependency graph for this program is shown in Figure 2.6. As shown, this program is not stratified since there is a cyclic path in the rule dependency graph that traverses an aggregation. Intuitively, we need to derive all `path` tuples before we can identify the one that is of minimum cost. Yet, `path` derivations are based on what "currently" exists in the `shortestPath` relation. As a result, the Datalog program in Figure 2.5 is not stratified.

It is however locally stratified. Assume that this program is evaluated using the seminaïve algorithm (e.g., Figure 2.2). The only option in the first step of the bottom-up evaluation is to derive all paths of length 1 using rule `r1`. Subsequent steps recursively use rule `r2` to derive paths of length $2, 3, \ldots$ (fully, in that order) until no further paths exist. These derivations are monotonic because we are performing a *min* aggregation of a *sum* over non-negative integers. As a result, rule evaluations derive `path` tuples of length k before path tuples of length $j > k$, which ensures that new `path` tuples are derived from a (seemingly) complete set of `shortestPath` tuples.

Many of the programs described in this thesis are not stratified, and of those, all are locally stratified. For example, the System R rules presented in Chapter 5 performs a *min* aggregation on the cost [5] of a query plan. This is used to select the "best plan" among the set of equivalent plans in a given level (plan size) of the System R dynamic program. The "best plan" is then recursively used to construct new plans, containing an extra predicate, for the next dynamic programming level. [6] Since adding an extra predicate to a query plan can only increase its cost (*principle of optimality*), and we fully explore all plans in a given level before moving to the next, this optimization is locally stratified.

[5] A non-negative integer value.

[6] Each level of the System R dynamic program adds an extra predicate to the query plan.

```
materialize(link, infinity, infinity, keys(1,2)).
materialize(path, infinity, infinity, keys(1,2,3)).
materialize(shortestPath, infinity, infinity, keys(1,2,3)).

link(''localhost:10000'', ''localhost:10001'', 1).
link(''localhost:10001'', ''localhost:10002'', 1).

r1 path(@X, Y, P, C) :-
   link(@X, Y, C), P := f_cons(X, Y).

r2 path(@X, Z, P, C) :-
   link(@X, Y, C1), path(@Y, Z, P2, C2),
   f_contains(X,P2) == false,
   P := f_cons(X,P2), C := C1 + C2.

r3 minCostPath(@X, Y, a_min<C>) :-
   path(@X, Y, _, C).

r4 shortestPath(@X, Y, P, C) :-
   minCostPath(@X, Y, C),
   path(@X, Y, P, C).

query shortestPath(''localhost:10000'', Y, P, C).
```

Figure 2.7: Shortest path program in Overlog. We follow the notation of Loo et al. [61]: `a_` prefixes introduce aggregate functions and `f_` prefixes introduce built-in functions. Variables that do not contribute to the rule evaluation are ignored using an underscore e.g., rule `r3`, third `path` attribute. We will use '...' to indicate a series of ignored variables.

2.2 Overlog: Our first look

Overlog marks a new beginning for the Datalog recursive query language, where distribution through data partitioning takes center stage. Like Datalog, an Overlog *program* consists of a set of deduction *rules* that define the set of tuples that can be derived from a base set of tuples called *facts*. Each rule has a *body* on the right of the `:-` divider, and a *head* on the left; the head represents tuples that can be derived from the body. The body is a comma-separated list of *terms*; a term is either a *predicate* (i.e., a relation), a *condition* (i.e., a relational selection) or an *assignment*. [7] An example Overlog program is shown in Figure 2.7. Overlog introduces some notable extensions to Datalog, which we describe before presenting the P2 runtime.

[7] Overlog's assignments are strictly syntactic replacements of variables with expressions; they are akin to "#define" macros in C++.

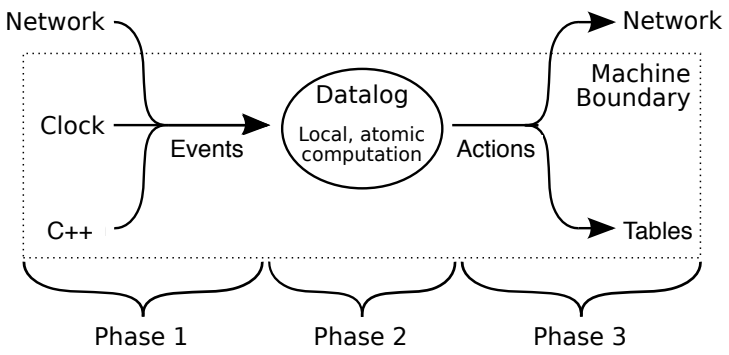


Figure 2.8: A single Overlog fixpoint.

2.2.1 Horizontal partitioning

Overlog's basic data model consists of relational tables that are partitioned across distributed nodes in a network. Each relation in an Overlog rule must have one attribute, whose variable is preceded by an "@" sign. This attribute is called the *location specifier* of the relation, and must contain values in the network's underlying address space (e.g., IP addresses for Internet settings, 802.13.4 addresses for sensor networks, hash-identifiers for code written atop distributed hash tables, etc.). Location specifiers define the horizontal partitioning of the relation: each tuple is stored at the address found in its location specifier attribute. At a given node, we call a tuple a *local tuple* if its location specifier is equal to the local address. Network communication is implicit in Overlog: tuples must be stored at the address in their location specifier, and hence the runtime engine has to send some of its derived tuples across the network to achieve this physical constraint. Loo, et al. provide syntactic tests to ensure that a set of rules can be maintained partitioned in a manner consistent with its location specifiers and network topology [62].

2.2.2 Soft State and Events

The three phases shown in Figure 2.8 describe a single evaluation round of an Overlog program. The input to this evaluation is a set of *event* tuples that are created when the network receives a packet, the system clock advances to some significant value [8], or through some arbitrary C++ code that updates the database. These events are queued in the first phase of the evaluation. An evaluator loop dequeues some number of these events and atomically executes a Datalog iteration. The rule deductions take the form of actions, which, in the third phase, cause data to be sent over the

[8]The Overlog language allows for the definition of a stream that periodically (based on real-time) produces a tuple with a unique identifier.

network or perform updates to the local database. These three phases represent a single time-step in the Overlog language.

Associated with each Overlog table is a "soft-state" lifetime that determines how long (in seconds) a tuple in that table remains stored before it is automatically deleted. Lifetimes can vary from zero to ∞. Zero-lifetime tables are referred to as *event* tables, and their tuples are called *events*; all other tables are referred to as *materialized* tables. An event only exists in the time-step that derived it, while materialized tuples span multiple time-steps, until explicitly deleted or when the lifetime expires (checked at the end of every time-step).

Overlog contains a `materialize` declaration that specifies the lifetime of a materialized table. At any time-step instance, at any given node in the network, the contents of the local Overlog "database" are considered to be: (a) the local tuples in materialized tables whose lifetime has not run out, (b) at most one local event fact across *all* event tables, and (c) any derived local tuples that can be deduced from (a) and (b) via one or more iterations of the program rules. Note that while (b) specifies that only one event fact is considered to be live at a time per node, (c) could include *derived* local events, which are considered to be live simultaneously with the event fact. This three-part definition defines the semantics of an Overlog program at a "snapshot in time." Overlog has no defined semantics across "time" and space (in the network); we describe the relevant operational semantics of the prototype in Chapter 2.3.

2.2.3 Deletions and Updates

Overlog, like SQL, supports declarative expressions that identify tuples to be deleted, in a deferred manner after a fixed point is achieved. To this end, any Overlog rule in a program can be prefaced by the keyword `delete`. In each timestep, the program is run to fixpoint, after which the tuples derived in `delete` rules – as well as other tuples derivable from those – are removed from materialized tables before another fixpoint is executed. It is also possible in Overlog to specify updates, but the syntax for doing so is different. Overlog's `materialize` statement supports the specification of a primary key for each relation. Any derived tuple that matches an existing tuple on the primary key is intended to *replace* that existing tuple, but this replacement happens through an insertion and a deletion: the deduction of the new tuple to be inserted is visible within the current fixpoint, whereas the deletion of the original tuple is deferred until after the fixpoint is computed.

2.2.4 A Canonical Example

To illustrate the specifics of Overlog, we describe the shortest paths example in Figure 2.7, which is similar to that of [62], but with fully-realized Overlog syntax that runs in P2. The three `materialize` statements specify that `link`, `path` and `bestpath`

```
r2a link_copy(@Y, X, Y, C1) :-
    link(@X, Y, C1).

r2b path(@X, Z, P, C) :-
    link_copy(@Y, X, Y, C1),
    path(@Y, Z, P2, C2),
    f_contains(X, P2) == false,
    P := cons(X, P2), C := C1 + C2.
```

Figure 2.9: The localized version of rule `r2` in Figure 2.7.

are all tables with ∞ lifetime and ∞ storage space. [9] For each table, the positions of the primary key attributes are noted as well. Rule `r1` can be read as saying "if there is a link tuple of the form `(X,Y,C)` stored at node `X`, then one can derive the existence of a path tuple `(X,Y,P,C)` at node `X`, where `P` is the output of the function `f_cons(X,Y)` – the concatenation of `X` and `Y`." Note that rule `r1` has the same location specifiers throughout, and involves no communication. This is not true of the recursive rule `r2`, which connects any `link` tuple at a node `X` with any path tuple at a neighboring node `Y`, the output of which is to be stored back at `X`. Figure 2.9 shows a rewritten version of rule `r2` [10], wherein all rule body predicates have the same location specifier; the only communication then is shipping the results of the deduction to the head relation's location specifier. Further details regarding the steps that perform this rule "rewrite" are presented in Chapter 3.3.

2.3 The P2 Runtime Engine

While ostensibly a network protocol engine, architecturally P2 resembles a fairly traditional shared-nothing parallel query processor, targeted at both stored state and data streams. The P2 runtime at each node consists of a compiler — which parses programs, optimizes them, and physically plans them — a dataflow executor, and access methods. Each P2 node runs the same query engine, and, by default, participates equally in every "query." In parallel programming terms, P2 encourages a Single-Program-Multiple-Data (SPMD) style for parallel tasks, but also supports more loosely-coupled (MPMD) styles for cooperative distributed tasks, e.g. for communications among clients and servers.

The P2 runtime is a dataflow engine that was based on ideas from relational databases and network routers; its scheduling and data hand-off closely resemble the Click extensible router [54]. Like Click, the P2 runtime supports dataflow *elements* (or "operators") of two sorts: pull-based elements akin to database iterators [37], and

[9] The third argument of P2's table definition optionally specifies a constraint on the number of tuples guaranteed to be allowed in the relation. The P2 runtime replaces tuples in "full" tables as needed during execution; replaced tuples are handled in the same way as tuples displaced due to primary-key overwrite.

[10] The new "localized" rules would replace the original rule `r2`.

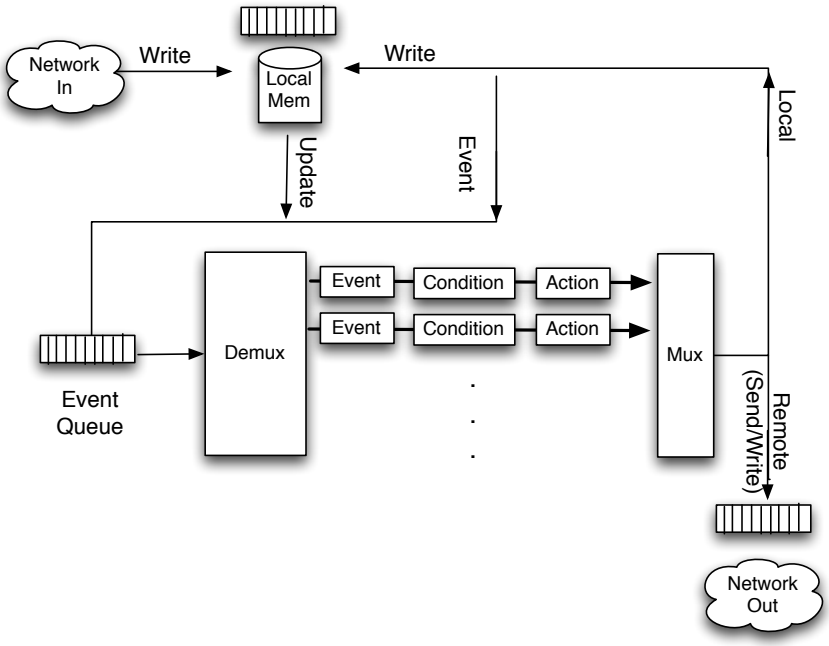


Figure 2.10: P2 Dataflow Architecture.

push-based elements as well. As in Click, whenever a pull-based element and a push-based element need to be connected, an explicit "glue" element (either a pull-to-push driver, or a queue element) serves to bridge the two. More details of this dataflow coordination are presented in the original P2 paper [63]. In Chapter 2.3.1 we describe the aspects of the dataflow architecture that affect our language semantics, and in Chapter 2.3.2 we describe the individual processing elements.

2.3.1 Dataflow Architecture

The P2 architecture consists of a dataflow of processing elements and queues, and a single driver loop. Figure 2.10 provides a high-level view (driver omitted) of this architecture, which contains three queuing elements. The `event` queue represents the primary input queue, which contains the current snapshot of tuples that the system uses to drive the processing. The `localmem` secondary queue feeds the main event queue with tuples when none currently exist in it. Tuples in the `localmem` queue represent side-affecting events (i.e., insert and delete) to local memory relations. P2 evaluates this queue in a tuple at a time fashion, where a single tuple is dequeued and executed in a "dataflow fixpoint."

The P2 architecture contains three output queues that hold the tuples derived from the rule engine. The choice of which queue a tuple is added to depends on the value of the location attribute. If the tuple's location is local to the current P2 instance, and its lifetime is greater than zero, then it will be added to the `localmem`

queue. If the tuple is remote, then it is added to the `netout` queue. The third output queue is the `event` queue. All (possibly many) local tuples that have a zero-lifetime are directly added to the event queue, which continues to drive rule deductions until no zero-lifetime tuples locally exist. This implementation decision exhibits a kind of "mini-fixpoint" (a side-effect unique to P2-Overlog) that we refer to as a *dataflow fixpoint*, which occurs when all tuples in the event queue have been drained. We describe this by example.

Assume a single tuple in the `localmem` input queue, and all other queues are empty. When the driver executes its "pull-push" element on the input of the empty `event` queue, it will dequeue a tuple in the `localmem` queue and add it to the `event` queue. In an iterative loop, the driver will dequeue a single tuple from the `event` queue and "route it" to the processing elements, which then produce some number of new tuple deductions. If any of those deductions contain local tuples with a zero lifetime, then they are reinserted into the `event` queue. The tuples with non-zero lifetimes represent "write" (insert or delete) actions against the local database (in `localmem`), and they are (silently) queued while the driver then continues to process tuples solely from the `event` queue until it is again empty. At this point, P2 declares a dataflow fixpoint, which triggers a flush of the local [11] "insertion" action writes from the (silent) queue, generating some number of new event tuples that are added to the `localmem` queue. If no "insertion" action tuples exist then a flush of the "deletion" action tuples occurs, and the corresponding deletion events are added (directly) to the `event` queue, before the process repeats, this time treating deductions as further deletions.

After all insertion and deletion tuples have been processed by the initial `localmem` input tuple, the system declares a *global fixpoint*. At this point, the driver loop will flush all tuples in the `netout` queue, triggering a transfer of those tuples to the corresponding P2 instances spanning the network. The driver loop then returns to the `localmem` queue for the next tuple to process.

From this perspective, the P2 runtime looks quite a bit like an Event-Condition-Action (ECA) system with a dataflow underneath: *events* are generated by the system clock and network components, while *conditions* are checked via dataflow processing elements, and *actions* initiate outbound network messages and updates to the database. A driver loop continuously routes events from the event queue to the "conditions" via the `demux` element in Figure 2.10. The initial input to the driver loop is the single tuple at the head of the `localmem` queue. This sole tuple is the input to the "current" fixpoint. Next, we describe the elements that implement the *condition* and *action* processing logic.

2.3.2 Dataflow Elements

The set of elements provided in P2 includes a suite of operators familiar from relational query engines: selection, projection, and in-memory indexes. These operators are

[11] Write actions from the network input are buffered and applied at the end of a "global fixpoint."

strung together to implement the logical *condition* of the processing loop. P2 supports joins of two relations in a manner similar to the symmetric hash join: it takes an arriving tuple from one relation, inserts it into an in-memory table for that relation, and probes for matches in an access method over the other relation (either an index or a scan). The work described in Chapter 3 extended this suite to include sorting and merge-joins, which allowed us to explore some traditional query optimization opportunities and trade-offs (Chapter 5).

P2 consists of exactly two logical *actions*: a local database write and a network send. We first describe the details behind a database write. An `event` tuple is modeled as transient database write, and therefore its action is the reinsertion into the `event` queue. P2 did not have support for persistent storage, beyond the ability to read input streams from comma-separated-value files. Its tables are stored in memory-based balanced trees that are instantiated at program startup; additional such trees are constructed by the planner as secondary indexes to support predicate join attributes. A write to the database is applied to the memory-based table, and a relevant (insert/delete) `event` is enqueued into the `localmem` queue.

The action for a remote output tuple is to simply enqueue it on the `netout` queue. When this queue is eventually flushed by the driver loop, all tuples in it are sent over the network prior to the next fixpoint iteration. As part of the same dataflow, P2 provides a number of elements used for networking, which handle issues like packet fragmentation and assembly, congestion control, multiplexing and demultiplexing, and so on; these are composable in ways that are of interest to network protocol designers [27]. The basic pattern that the reader should assume is that each P2 node has a single IP port for communication, and the dataflow graph is "wrapped" in elements that handle network ingress with translation of packets into tuples, and network egress with translation of tuples into packets.

Chapter 3

Evita Raced: Metacompiler

Declarative Networking has the potential to expand the lessons and impact of database technologies into new domains, while reviving interest in classical database topics like recursive query processing that have received minimal attention in recent years. Yet our own system was entirely implemented in an imperative programming language: the initial version of the P2 runtime was implemented in C++ [63]. We asked ourselves whether Codd's vision applies to our own efforts: can declarative programming improve the implementation of declarative systems?

In this chapter, we put declarative systems "in the mirror" by investigating a declarative implementation of one key component in any relational database system, the query compiler. Specifically, we reimplemented the query compiler of P2 as a *metacompiler*: a compiler (optimizer) for the P2 language, Overlog, that is itself written in Overlog. We named the resulting implementation "Evita Raced."[1] Using Evita Raced, we extended P2 with a number of important query optimization techniques it formerly lacked, and found that our declarative infrastructure made this quite elegant and compact.

The elegance of our approach was derived in part from the fact that many query optimization techniques – like many search algorithms – are at heart recursive algorithms, and therefore would benefit from a declarative approach in much the same way as networking protocols. Even non-recursive optimization logic – such as parts of the magic-sets algorithm presented in Chapter 4 – are simple enough to express in a declarative fashion that abstracts away mechanistic details such as state cleanup (e.g., garbage collection) and invariant enforcement via key constraints and materialized view maintenance.

The remainder of this chapter is organized as follows. We describe the architecture of Evita Raced in Chapter 3.1, which involves compiling an Overlog program into a relational representation. Compiling code into data is necessary in order to then express compilation steps (i.e., rewrites, optimizations) as queries. In Chapter 3.1.1,

[1] "Evita Raced" is almost "Declarative" in the mirror, but as with the Overlog language itself, it makes some compromises on complete declarativity.

we describe the schema of the compiled code, which is packaged in a Metacompiler Catalog. The architecture of Evita Raced is described in Chapter 3.1.2 as a dataflow of compilation steps and a scheduler to determine compilation step order. A given compilation step is called a *stage*, which can be written in either C++ or Overlog. Chapter 3.1.3 describes our four basic C++ stages that bootstrap the compiler into a state that permits the subsequent dynamic installation of Overlog stages. In Chapter 3.2, we present our first declarative compilation stage: the delta rewrite [62] for rewriting a rule into a form suitable for seminaïve evaluation. In Chapter 3.3, we describe our Overlog rules for expressing the localization rewrite [63], which rewrites distributed (join) rules into a locally executable form. Chapter 3.4 contains some final thoughts on the Evita Raced architecture. Further declarative stages are then presented in Chapter 4 (magic-sets rewrite) and Chapter 5 (System R and Cascades cost-based optimizations).

3.1 Declarative Compilation

Evita Raced is a compiler (i.e., query optimizer) for the Overlog declarative language that supports a runtime-extensible set of program rewrites and optimizations, which are themselves expressed in Overlog. This metacompilation approach is achieved by implementing optimization logic via dataflow programs (query plans) running over a table representation of the compiler state. Two main challenges must be addressed to make this work. First, all compiler state – including the internal representation of both declarative Overlog programs and imperative dataflow programs – must be captured in a relational representation so that it can be referenced and manipulated from Overlog. Second, the (extensible) set of tasks involved in optimization must itself be coordinated via a single dataflow program that can be executed by the P2 runtime engine. In this chapter, we describe the implementation of the Evita Raced framework, including the schema of the compiler state, the basic structure of the Evita Raced dataflow graph, and the basic dataflow components needed to bootstrap the architecture.

3.1.1 Table-izing Optimizer State

A typical query optimizer maintains a number of data structures to describe the contents of a query, and to represent the ongoing state of a query planning algorithm, including fragments (i.e., subplans) of query plans. Our first task in designing Evita Raced was to capture this information in a relational schema.

Figure 3.1 shows an Entity-Relationship (ER) diagram we developed that captures the properties of an Overlog program, and its associated P2 dataflow query plans. In the figure, entities are *squares* with attributes hanging off of them as *ovals*. An attribute has a name, and if it is part of the primary key, then it is shown in bold.

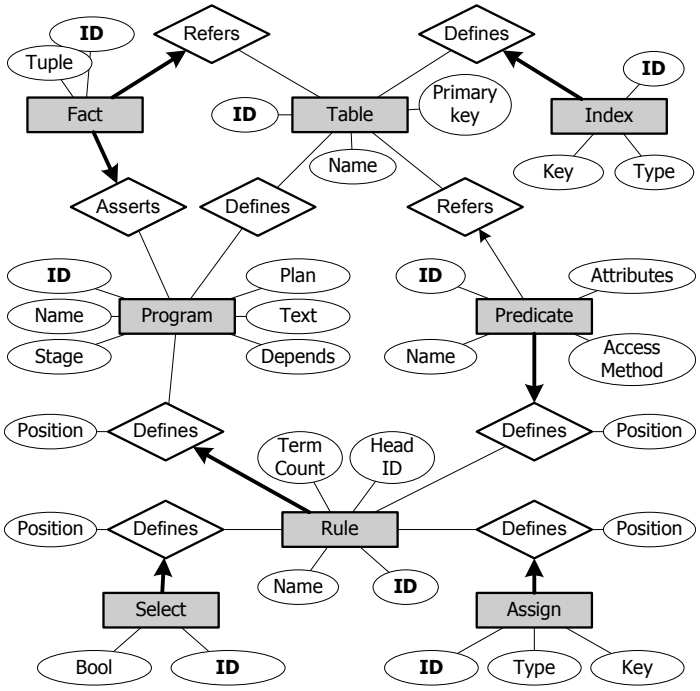


Figure 3.1: ER Diagram of a query plan in P2.

Relationships are shown as *diamonds* that include a name description. Lines connect entities to relationships and identify the following constraints.

- A bold line indicates the existence of at least one tuple in the output of a "foreign-key join" with the connecting entities, while a regular line imposes no constraints on the "join" output.
- An arrow directed into a relationship indicates *many* tuples from the origin entity "join with" exactly *one* tuple from the entity on the other side of the relationship.

We derived the constraints in the diagram by reviewing the semantic analysis rules enforced in the original P2 compiler; we discuss a few of them here for illustration. An Overlog *rule* must appear in exactly one *program*. A *select* term (e.g., `f_contains(X,P2) == false` in Figure 2.7) is a Boolean expression over attributes in the predicates of the rule, and must appear in exactly one *rule*. The diagram indicates that a *predicate* must also appear in a unique *rule*, and that it may possibly reference a single *table*. A predicate that references a table is called a *table predicate* (or a *materialized predicate*), while one that does not reference a table is called an *event predicate*. An *index* is defined over exactly one *table*, and a *table* defines at least one index (namely the primary key index, which P2 always constructs). Some relations may contain *facts* (input tuples) at startup, each of which must belong to a single program and must reference a single table.

Name	*Description*	*Relevant attributes*
table	Table definitions	**table_id**, primary_key
index	Index definitions	**index_id**, **table_id**, keys, type
fact	Fact definitions	**program_id**, **table_id**, **id**, tuple
program	User program description	**program_id**, name, stage, text, depends, plan
rule	Rules appearing in a program	**program_id**, **rule_id**, name, term_count, head_id
predicate	Relational predicates	**id**, **rule_id**, table_id, name, position, access_method
select	Selection predicates	**id**, **rule_id**, boolean, position
assign	Variable substitution statements	**id**, **rule_id**, variable, value, position

Figure 3.2: The Metacompiler Catalog: tables defining an Overlog program and dataflow execution plan. The primary key columns are shown in bold.

The conversion from ER diagram to relational format was a textbook exercise [77]. Table 3.2 lists the set of relations that capture the entities mentioned in the ER diagram; we refer to this as the *Metacompiler Catalog*. We modified P2 to create these tables at system startup, and they are accessible to any system-authorized Overlog programs (i.e., optimizations) added to the system.

3.1.2 Metacompiler Architecture

Optimization logic expressed in Overlog is declarative, and Evita Raced realizes this logic by converting it to a dataflow program to be executed by the P2 dataflow subsystem (Chapter 2.3). Here, we describe how Evita Raced represents query optimization programs as dataflow, and also the way it orchestrates multiple different optimization programs in P2.

An optimizer built using Evita Raced is composed of an extensible number of *stages*, each of which performs some compilation task on the input program. Table 3.3 describes the primary compiler stages packaged with the Evita Raced framework. An Evita Raced stage can be written as a dataflow program of one or more P2 elements in C++, which are then compiled into the P2 binary; this is how we implement certain base stages required for bootstrapping (Chapter 3.1.3). However, the power of Evita Raced comes from its support for stages written in Overlog. In addition to being compactly expressed in a high-level language, Overlog stages can be loaded into a running P2 installation at any time, without the need to compile a new P2 binary.

A stage programmer registers a new stage with Evita Raced by inserting a tuple into the `program` relation. Such a tuple contains an unique identifier (*program_id*), a name (*name*), a list of stage dependencies (*depends* — Chapter 3.1.2), and the program text (*text*). The `program` relation also contains an attribute for the name of

Stage name	*Language*	*Description*
StageScheduler (Chapter 3.1.2)	C++	Coordinates the compilation of stages.
Parser (Chapter 3.1.3)	C++	Bison based parser engineered to populate the Metacompiler Catalog with data from the program AST.
Planner (Chapter 3.1.3)	C++	Generates a dataflow description from the program data contained in the Metacompiler Catalog.
Installer (Chapter 3.1.3)	C++	Instantiates C++ dataflow objects from a dataflow description.
Delta Rewrite (Chapter 3.2)	Overlog	Converts rules based on materialized tables into an ECA form.
Localization (Chapter 3.3)	Overlog	Rewrites distributed (join) rules into a locally executable form.
Magic-sets (Chapter 4)	Overlog	Rewrites rules to include magic predicates, which act as selection predicates for constants contained in query predicates.
System R (Chapter 5.2)	Overlog	A *top-down* dynamic programming optimization.
Cascades (Chapter 5.3)	Overlog	A *bottom-up* dynamic programming optimization.

Figure 3.3: Primary Evita Raced compiler stages.

the compiler stage currently operating on the program (*stage*), and the final physical plan (*plan* — Chapter 3.1.3); these attributes are used to convey partial compilation results from stage to stage. We next describe the interfaces to an Evita Raced compiler stage and how we schedule different stages when compiling (any) Overlog programs.

The Stage API

An Evita Raced stage can be thought of as a stream query that listens for a tuple to arrive on an event stream called `<stage>::programEvent`, where `<stage>` is the name of the stage. The `<stage>::programEvent` table contains all the attributes mentioned in the `program` table. When such a tuple arrives, the queries that make up that stage execute, typically by modifying catalog tables in some way. When a stage competes it inserts a new `program` tuple, including the current stage name in the *stage* attribute, into the program table.

To represent this behavior in a stage written in Overlog, a relatively simple template can be followed. An Overlog stage must have at least one rule body containing the `<stage>::programEvent` predicate. These stage *initiation* rules react to new programs arriving at the system and trigger other rules that are part of the same stage. In addition, the stage must have at least one rule that inserts a `program` tuple into the `program` table to signal its completion.

Stage Scheduling

In many cases, optimization stages need to be ordered in a particular way for compilation to succeed. For example, a *Parser* stage must run before any other stages, in order to populate the Metacompiler Catalog. The *Planner* must follow any query transformation stages, since it is responsible for translating the (relational) logical query plan into a physical dataflow representation. And finally, the *Installer* stage must follow the *Planner*, since it instantiates dataflow specifications as P2 C++ elements, and installs them into the P2 runtime.

A natural way to achieve such an ordering would be to "wire up" stages explicitly so that predecessor stages directly produce `<stage>::programEvent` tuples for their successors, in an explicit chain of stages. However, it is awkward to modify such an explicit dataflow configuration upon registration of new stages or precedence constraints. Instead, Evita Raced captures precedence constraints as *data* within a materialized relation called `StageLattice`, which represents an order (i.e., an acyclic binary relation) among stages; this partial order is intended to be a lattice, with the *Parser* as the source, and the dataflow *Installer* as the sink.

To achieve the dataflow connections among stages, the built-in *StageScheduler* component (itself a stage) listens for updates to the `program` table, indicating the arrival of a new Overlog program or the completion of a compiler stage for an on-going program compilation. The *StageScheduler* is responsible for shepherding compilation stage execution according to the `StageLattice`. Given a `program` update, the StageScheduler "joins with" the `StageLattice` to identify a next stage that can be invoked, and derives a `<stage>::programEvent` tuple that will start the given stage; the contents (attributes) of the `<stage>::programEvent` tuple are the same as those in the updated `program` tuple.

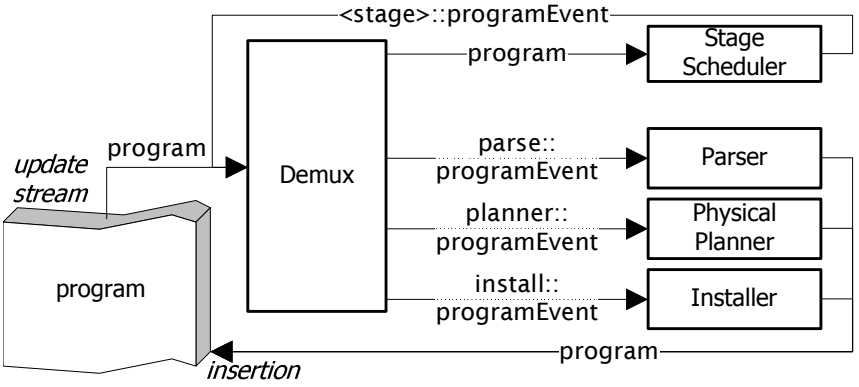


Figure 3.4: The Evita Raced (cyclic) dataflow architecture, containing only the default compilation stages. The arrows leaving the Demux element route tuples, based on the tuple name, to the relevant stages on the right. We focus here on the portion of the P2 dataflow that corresponds to the Evita Raced architecture.

The StageScheduler and any compilation stages (whether built-in or runtime-

installed) are interconnected via the simplified dataflow illustrated in Figure 3.4. The Evita Raced architecture is embedded in the same P2 dataflow used to execute user queries. As described in Chapter 2.3 (and [63]), the dataflow consists of a C++ "demultiplexer" that routes tuples from its input (on the left) to individual event handlers listening for particular tuple names. The Evita Raced runtime simply adds these "default stages" to the bootstrap routine of the P2 system.

Consider the simplicity of how Evita Raced architecture coexists with the P2 dataflow. To install a new (Overlog) compilation stage into the runtime, the Installer stage (Chapter 3.1.3) simply extends the *Demux* element to include a port for `<stage>::programEvent` tuples, routing them to the respective rule(s) of a given stage's Overlog program. The `StageLattice` relation is also updated (e.g., through fact tuples in the Overlog stage program) to include its position in the compilation pipeline. Once installed, the Overlog stage need only follow a simple protocol for when and how it should execute.

The protocol, followed by stages, indicates when a stage should start (after receiving a `<stage>::programEvent` tuple) and what it must do on completion. When a stage completes, the only requirement is to update the `program` table with the "stage" attribute set to the current stage name. The *StageScheduler* receives all such updates to the `program` table – see Figure 3.4, the *Demux* `program` tuple port into the *StageScheduler* – and uses the value of the `program` *depends* attribute along with the `StageLattice` relation to determine the next stage. This covers the full Evita Raced compilation process of an Overlog program, from the *Parser* stage to the *Installer* stage, and any other stages along the way.

To sum up, the lifecycle of a program compilation starts when a user submits a `program` tuple to the system with a `null` stage attribute. The StageScheduler receives that `program` tuple and generates a `parse::programEvent` tuple (the Parser being the source stage in the lattice), which is routed by the Demux element to the Parser stage. When the Parser is done, it updates that `program` tuple in the corresponding table, changing the tuple's *stage* attribute to "Parser." The StageScheduler receives the `program` tuple, and routes a `planner::programEvent` to the Demux and eventually the Planner, which goes round the loop again to the Installer. Finally, once the Installer is done and notifies the StageScheduler via a `program` tuple with the *stage* attribute set to "Installer," the StageScheduler concludes the compilation process. If the Overlog program being parsed is itself a new compilation stage, then after installation, the scheduler updates the stage lattice (e.g., by applying stage lattice facts defined in the stage program).

3.1.3 Compiler Bootstrapping

This section describes the baseline Evita Raced compiler as four simple C++ stages that are loaded by the P2 bootstrap routine. As in many metaprogramming settings, this is done by writing a small bootstrap component in a lower-level language. Evita Raced is initialized by a small C++ library that constructs the cyclic dataflow of Fig-

ure 3.4, including the four default stages shown. The bootstrap compiler is sufficient to compile simplified Overlog programs (local rules only, no optimizations) into operational P2 dataflows. We describe here the implementation of our Parser, Planner, and Installer bootstrap stage elements, which form the core foundation of the Evita Raced architecture.

Parser

The Parser passes the program text it receives in the `parse::programEvent` through a traditional lexer/parser library specified using flex [2] and bison[1]; this library code returns a standard *abstract syntax tree* representation of the text. Assuming the Parser does not raise an exception due to a syntax error, it walks the abstract syntax tree, generating Metacompiler Catalog tuples for each of the semantic elements of the tree. In addition to recognizing the different terms of each rule, the parser also annotates each term with a position, relative to its "parse" order. In Chapter 3.2, we will use this position when "compiling" a rule into ECA form, and in Chapter 5, we use it to reorder subgoals in the rule body for optimizing the join order.

Physical Planner

The Planner stage is responsible for doing a naïve translation of Metacompiler Catalog tuples (i.e., a parsed Overlog program) into a dataflow program. It essentially takes each rule and deterministically translates it into a dataflow graph language, based on the rule term positions.

More specifically, for each rule, the Planner considers each term (predicate, selection or assignment) in order of the position attribute contained in the relevant Metacompiler Catalog relation. The predicate representing the event stream is always planned first, and registers a listener in the Demux element (recall Figure 3.4). The terms following the event stream are translated, left-to-right, into a C++ dataflow in the same way that the original P2 system did using select-project-join operator methods.

We further mention three specific details. First, where the original P2 system translated a logical query plan directly to a software dataflow structure in C++, we have chosen to create an intermediate, textual representation of the dataflow. This representation is in a language akin to the Click router's dataflow language, but we omit its details here.

Second, unlike the original P2 system, we have introduced a number of new join methods for in-memory tables. Prior to this work, P2 only supported index-nested-loop joins, where the appropriate index was built on the join column(s) during program compilation. We have added two elements to the P2 runtime that perform a simple nested-loop join and a sort-merge join on a tuple from the outer input, with a relation on the inner. We note that our sort-merge join is not traditional: it only

requires the inner relation to be sorted. The P2 architecture was not optimized for blocking operators; a consequence of its tuple at a time dataflow evaluator (Chapter 2.3.1). Therefore, we decided not to sort the outer relation in a sort-merge join, and instead perform a binary search on the inner relation for each outer (streaming) tuple. The `predicate` relation contains the choice of join method as one of its attributes, and the Planner creates the appropriate dataflow element that implements the given join method.

Third, the Planner only understands rules that are in an event-condition-action (ECA) form. An Overlog rule may have no event predicate (e.g., "`table1 :- table2, table3.`"). A *delta rewrite* (from Loo, et al. [62]) is used to convert such rules in an ECA form (E.g., "`table1 :- delta_table2, table3.`" and "`table1 :- table2, delta_table3.`".) As in earlier versions of P2 [62], `delta_table` denotes a stream conveying insertions, deletions, or timeout refreshes to tuples of the target `table`. We could have done this directly in the Planner, but instead we built it as an Overlog stage (Chapter 3.2). This decision had an important consequence; when expressing the delta rewrite stage in Overlog, we had to use rules that contained an explicit event predicate. Furthermore, any Overlog program that contains rules with no explicit event predicate, depends on the delta rewrite stage. The delta rewrite stage consists of a mere 12 Overlog (ECA) rules (25 lines of code), and is one of the first Overlog stages to be compiled into the system.

Installer

Following the Planner stage, what remains is to parse the textual representation of the physical dataflow, create the corresponding C++ elements, and "wire them up" accordingly. We have implemented these steps in a single C++ Installer stage. Once the elements and their connections are instantiated, the Installer stage stitches them into the P2 runtime's overall dataflow graph. In other words, the Installer implemented an extensible dataflow runtime by dynamically adding new rule instantiations to (possibly new) Demux ports (see Figure 3.4); a feature not available prior to the release of Evita Raced.

3.1.4 Modularity

A stage adds a weak notion of modularity to the Overlog language. Prior to Evita Raced, P2 was only able to install a single Overlog program into its dataflow. The rules in this program had complete visibility of all materialized relations, and accordingly side effects to these relations were visible throughout. In this work, we had to ensure that side effects made by one stage were not visible in another, since such overlapping updates against the Metacompiler Catalog could render it inconsistent.

The first Overlog stage installed into Evita Raced adds stage modularity to the system. Itself a rewrite, the stage adds *guard* predicates to all rule bodies in subsequently installed Overlog stages. These guard predicates ensure that only rules in an

"active" stage react to Metacompiler Catalog updates. A stage is activated when its `<stage>::programEvent` tuple is first derived, and deactivated when the stage inserts a finalized `program` tuple. Facts added to the `guard` relation activate the rules of a stage, and deactivate rules in other stages. We do not mention these guard rules further since they are completely abstracted way from the programmer.

3.1.5 Discussion

The metacompilation approach of Evita Raced led us to naturally design the system extensibility around issues of data storage and dataflow, rather than library loading and control flow modifications. While rule-based systems are usually intended to be easier to extend than a procedural system, the internal implementation of Evita Raced is clean, due to our thorough embrace of the native dataflow infrastructure, which we use both to execute optimization code, and orchestrate stages via precedence tables and the StageScheduler cycle. The result of this design is that even a major addition to the Evita Raced compiler entails very minimal modification to the runtime state: only the addition of a pair of dataflow edges to connect up the new stage, and the insertion of precedence tuples in a single table. Beyond the StageScheduler and the four bootstrap stages, no additional extensibility code was added to P2 to support Evita Raced.

Despite its simplicity, Evita Raced is flexible enough that other researchers have used it to enhance P2 with support for new languages at both its input and output. First, by extending the Parser element and registering some Overlog rules, Abadi and Loo were able to get P2 to optimize and rewrite programs written in a new language, which extends Overlog with the ability to attest to the provenance of data [5]. Second, Chu, et al. were able to use Evita Raced to cross-compile Overlog programs into dataflow specifications that execute on the DSN platform, a declarative networking system that runs on wireless sensor nodes [24].

3.2 The Delta Rewrite

In this section we describe our declarative implementation of the delta rewrite for Overlog rules. The rewrite itself consists of only 12 rules, which includes rules for stage activation, finalization and general housekeeping of the Metacompiler Catalog relations. Before diving into the specific rules for the rewrite, we describe its actions by example.

3.2.1 Delta by Example

Consider the shortest path program in Figure 3.5, copied over from Figure 2.7. First thing to notice is the **materialize** statements at the top. They indicate that posi-

```
materialize(link, infinity, infinity, keys(1,2)).
materialize(path, infinity, infinity, keys(1,2,3)).
materialize(shortestPath, infinity, infinity, keys(1,2,3)).

r1 path(@X, Y, P, C) :-
   link(@X, Y, C), P := f_cons(X, Y).

r2 path(@X, Z, P, C) :-
   link(@X, Y, C1),
   path(@Y, Z, P2, C2),
   f_contains(X, P2) == false,
   P := f_cons(X, P2), C := C1 + C2.

r3 minCostPath(@X, Y, a_min<C>) :-
   path(@X, Y, P, C).

r4 shortestPath(@X, Y, P, C) :-
   minCostPath(@X, Y, C),
   path(@X, Y, P, C).
```

Figure 3.5: Shortest path program.

tive lifetime tables should exist for `link`, `path`, and `shortestPath` tuples, along with the appropriate primary key columns. Since there is no **materialize** statement for the `minCostPath` predicate, P2 considers such tuples to be events, that will end up triggering rule `r4` when "pulled" from the event queue. The question then is, what triggers the other rules to produce `minCostPath` event tuples?

The delta rewrite converts the rules in Figure 3.5 into the rules shown in Figure 3.6. The *new* rules contain a single Δ predicate, shown (by convention) at the front of the rule. Since rule `r4` already contains an *e*vent predicate, the delta rewrite simply ignores this rule, which is fixed to trigger off of `minCostPath` tuples. The remaining rules are converted into Δ form so that they can be installed into the P2 runtime by the *Planner* stage. We start with rule `r1`, which contains the single subgoal `link`. The delta rewrite simply adds a delta annotation to this predicate, informing the *Planner* to trigger the rule when a receive/insert/delete event occurs on the `link` relation. The same thing happens in rule `r3` w.r.t., the `plan` relation.

Rule `r2` has two subgoals `link` and `path`, both of which are materialized tables. In this case, we must break the rule into two disjoint rules (one for each materialized subgoal). The first of these rules will trigger on (say) the `link` tuple, followed by a join with `path`, etc. The second rule triggers on `path` event tuples, and joins with the `link` relation, etc. Both of these rules project onto the `path` relation, which in turn triggers further invocations of rule `r2b` on new `path` data.

```
materialize(link, infinity, infinity, keys(1,2)).
materialize(path, infinity, infinity, keys(1,2,3)).
materialize(shortestPath, infinity, infinity, keys(1,2,3)).

r1 path(@X, Y, P, C) :-
   Δlink(@X, Y, C),
   P := f_cons(X, Y).

r2a path(@X, Z, P, C) :-
    Δlink(@X, Y, C1),
    path(@Y, Z, P2, C2),
    f_contains(X, P2) == false,
    P := f_cons(X, P2), C := C1 + C2.

r2b path(@X, Z, P, C) :-
    Δpath(@Y, Z, P2, C2),
    link(@X, Y, C1),
    f_contains(X, P2) == false,
    P := f_cons(X, P2), C := C1 + C2.

r3 minCostPath(@X, Y, a_min<C>) :-
   Δpath(@X, Y, P, C).

r4 shortestPath(@X, Y, P, C) :-
   minCostPath(@X, Y, C),
   path(@X, Y, P, C).
```

Figure 3.6: Delta rewrite rules from Figure 3.5.

```
/*Initiate a rewrite at position 1 of a rule not already containing an event
predicate in this position. */
d1 rewrite(@A, Pid, Rid, PredID, f_idgen(), f_idgen(), Pos) :-
   delta::programEvent(@A, Pid, ...),
   sys::rule(@A, Rid, Pid, _, HeadID, _, _, _, Goals),
   sys::predicate(@A, PredID, Rid, _, Name, Tid, _, Schema, Pos),
   Tid != null, Pos == 1.

/*Initiate a rewrite position for each predicate in the rule body. */
d2 rewrite(@A, Pid, Rid, PredID, f_idgen(), f_idgen(), Pos) :-
   rewrite(@A, Pid, Rid, ...),
   sys::predicate(@A, PredID, Rid, ..., Schema, Pos),
   Pos > 1.
```

Figure 3.7: Deduce a rewrite fact for a new delta rule to be created for a particular table predicate in the original rule's body.

```
/*Put the delta predicate in the first position of the new rule. */
d3 sys::predicate(@A, f_idgen(), NewRid, Notin, Name, Tid, "DELTA",
                  Schema, 1) :-
   rewrite(@A, Pid, Rid, DeltaPredID, NewRid, NewHeadID, _),
   sys::predicate(@A, DeltaPredID, Rid, Notin, Name, Tid, ECA,
                  Schema, Pos).

/*Make a new head predicate for the new rule by copying the old head predicate. */
d4 sys::predicate(@A, NewHeadID, NewRid, Notin, Name, Tid, ECA,
                  Schema, 0) :-
   rewrite(@A, Pid, Rid, DeltaPredID, NewRid, NewHeadID, _),
   sys::rule(@A, Rid, Pid, _, HeadID, ...),
   sys::predicate(@A, HeadID, Rid, Notin, Name, Tid, ECA, Schema, _).
```

Figure 3.8: Rules that copy the old head predicate from the old rule to the new rule, and creates the delta predicate in the new rule from the subgoal referenced by the rewrite tuple.

3.2.2 Declarative Delta

We now turn to the delta rewrite Overlog stage; used translate Figure 3.5 into Figure 3.6. Prior to the installation of this stage, only rules containing an explicit event predicate can be installed. As a result, all rules described here contain an explicit event predicate e.g., the `delta::programEvent` tuple.

Figure 3.7 contains two rules that initiate the delta rewrite by deducing a rewrite tuple from each rule in the target program. Rule `d1` triggers on the `delta::programEvent` tuple and "joins with" the `rule` and `predicate` tables, selecting out the predicate in position 1. Recall that this is the event position, and that this rewrite ignores rules containing an explicit event. Therefore, if the predicate in position 1 references a materialized table ($Tid != null$) — it is not an event — then we need to rewrite it. Once a `rewrite` event tuple has been deduced for a given target rule, rule `d2` initiates a second `rewrite` event tuple for each predicate in the target rule's body. The $Pos > 1$ selection avoids the head predicate (position 0) and the first predicate already handled by rule `d1`.

Assume we have initiated a rewrite tuple for a given rule r and body predicate p_i at position i. The next step is to actually create the new rule Δr_i with predicate Δp_i that references the delta events of predicate p_i. The new rule will take the following form h `:-` $\Delta p_i, G_{j!=i}$, where h references the original head predicate in rule r and $G_{j!=i}$ is the list of subgoals that exclude predicate p_i.

Figure 3.8 contains two rules that create the head predicate (h) and the delta predicate (Δp_i) for the new delta rule (Δr_i). Rule `d3` specifically creates the delta predicate, placing it in position 1 (by convention) of the new rule. Next, rule `d4` copies the head predicate, from the old rule, by joining the old rule identifier (Rid) in `rewrite` with the `rule` table, and the `predicate` table along the old head predicate

```
/*Kick off an iterator for the remaining rule subgoals. */
d5 remainder(@A, Pid, Rid, NewRid, 1, 2, Pos) :-
   rewrite(@A, Pid, Rid, DeltaPredID, NewRid, _, Pos).

/*Forward the remainder iterator along the subgoals. */
d6 remainder(@A, Pid, Rid, NewRid, OldPos+1, NewPos, DeltaPos) :-
   remainder(@A, Pid, Rid, NewRid, OldPos, NewPos, DeltaPos),
   sys::rule(@A, Rid, Pid, ..., Goals),
   OldPos < Goals,
   NewPos := OldPos == DeltaPos ? NewPos : NewPos + 1.

/*Copy table predicate to the new delta rule. */
d7 sys::predicate(@A, f_idgen(), NewRid, Notin, Name, Tid, null,
                  Schema, NewPos) :-
   remainder(@A, Pid, Rid, NewRid, OldPos, NewPos, DeltaPos),
   sys::predicate(@A, PredID, Rid, Notin, Name, Tid, _, Schema, OldPos),
   OldPos != DeltaPos.

/*Make a new assignement for the new delta rule. */
d8 sys::assign(@A, f_idgen(), NewRid, Var, Value, NewPos) :-
   remainder(@A, Pid, Rid, NewRid, OldPos, NewPos, _),
   sys::assign(@A, Aid, Rid, Var, Value, OldPos).

/*Make a new selection for the new delta rule. */
d9 sys::select(@A, f_idgen(), NewRid, Bool, NewPos) :-
   remainder(@A, Pid, Rid, NewRid, OldPos, NewPos, _),
   sys::select(@A, Sid, Rid, Bool, OldPos, AM).
```

Figure 3.9: Rules that copy old subgoals to the new delta rule.

identifier ($HeadID$). The new head predicate is given a new predicate identifier ($NewHeadID$) and the new rule identifier ($NewRid$).

Figure 3.9 contains the next group of rules that copy the body predicates $G_{j!=i}$ from the old rule r to the new Δr_i rule, excluding predicate p_i. We express this through a secondary group of event tuples, called `remainder`, that reference each body predicate in rule r excluding p_i. A `remainder` tuple contains the predicate position relative to rule r and a new position in the Δr_i rule. The new position must start at 2, following the delta predicate, which is already set to p_i.

Rule `d5` initiates the first `remainder` tuple from a `rewrite` tuple, and rule `d6` carries further `remainder` deductions along each subgoal in the body of rule r. A `remainder` tuple contains three attributes that reference rule positions; shown here by the $OldPos$, $NewPos$ and $DeltaPos$ variable names. The $OldPos$ variable specifies the predicate position to copy from the original rule to the new Δ rule, and $NewPos$ is its position in this new rule. The $DeltaPos$ variable refers to the position of p_i, the new Δ predicate, in the original rule. Special logic is used to avoid predicate p_i w.r.t., `remainder` tuples. For example, rule `d6` does not increment the $NewPos$ when $OldPos == DeltaPos$. The remaining rules (`d7`, `d8` and `d9`) deal with copying

```
/*Create the new rule */
d10 sys::rule(@A, NewRid, Pid, Name, NewHeadID, null, Delete, Goals) :-
    rewrite(@A, Pid, Rid, DeltaPredID, NewRid, NewHeadID, Pos),
    sys::predicate(@A, DeltaPredID, Rid, _, PredName, ...),
    sys::rule(@A, Rid, Pid, RuleName, _, _, Delete, Goals),
    Name := RuleName + "delta" + PredName + Pos.

/*Clean up old rule state */
d11 delete sys::rule(@A, Rid, Pid, Name, HeadID, P2DL, Delete, Goals) :-
    rewrite(@A, Pid, Rid, ...),
    sys::rule(@A, Rid, Pid, Name, HeadID, P2DL, Delete, Goals).

/*Signal the completion of the delta rewrite to the StageScheduler. */
d12 sys::program(@A, Pid, Name, Rewrite, "delta", Text, Msg,
                 P2DL, Src) :-
    programEvent(@A, Pid, Name, Rewrite, Status, Text, Msg, P2DL, Src).
```

Figure 3.10: Creates a new rule tuple that references the delta rewrite rule. Cleans up the old (non-delta) rule. Inserts a program tuple indicating that the delta rewrite has finished.

```
r2a link_copy(@Y, X, Y, C1) :-
    link(@X, Y, C1).

r2b path(@X, Z, P, C) :-
    link_copy(@Y, X, Y, C1),
    path(@Y, Z, P2, C2),
    f_contains(X, P2) == false,
    P := f_cons(X, P2), C := C1 + C2.
```

Figure 3.11: The localized version of rule `r2` in Figure 3.5.

subgoals rule r to the new Δr_i rule. Notice that rule `d7` skips predicate p_i at position *DeltaPos*.

The final set of rules shown in Figure 3.10 perform housekeeping tasks related to this rewrite. Rule `d10` creates a rule tuple that references the delta rule Δr_i for a given delta predicate Δp_i. The old rule r is deleted in rule `d11`, which, through materialized view maintenance, removes the old head predicate and subgoals. Finally, rule `d12` inserts a `program` tuple that indicates the completion of the delta rewrite stage. [2]

3.3 The Localization Rewrite

We briefly describe the localization compiler stage, which turns a rule with multiple location specifiers in its body, into many rules, each of which has a single location

[2]Note that this entire rewrite is performed in a single P2 *dataflow* fixpoint.

specifier in its body; turning a distributed join into a set of local joins with partial result transmissions among the rules involved [62]. This rewrite was part of the original P2 system, but implemented in C++ and woven into the monolithic compiler. In Evita Raced, the localization rewrite stage contained 11 rules that resembled the rules in the delta rewrite stage. Therefore, we provide a high level description of this rewrite, and its declarative structure.

We start with an example description using rule `r2` from Figure 3.5. This rule is rewritten into the two rules shown in Figure 3.11. The `link_copy` (event) predicate forwards `link` tuples at node X to node Y. This will result in a network transfer of `link` tuples @X to `link_copy` tuples @Y. At node Y, the `link_copy` tuples trigger rule `r2b`, which completes the execution of rule `r2` before sending the `path` results back to node X.

Declaratively, the localization stage traverses distributed rules in left-to-right order; rules with local-only body predicates are selected out early in the stage. The location attribute of the current predicate in this traversal is stored along with the cursor information of the traversal. A `rewrite` is derived if the traversal reaches a predicate with a location attribute that differs from the previous. The `rewrite` tuple *splits* the rule at the given position, creating a new *glue* predicate IR_copy, and two new rules defined as follows.

1. **IR_copy** :- (predicates to the left, excluding the `rewrite` position).
2. (original rule head predicate) :- **IR_copy**, (predicates to the right, including the `rewrite` position).

The location attribute in the **IR_copy** predicate is taken from the predicate at the `rewrite` position. That is, the predicate with the "new" location attribute. The other attributes in the **IR_copy** predicate are taken from the predicates to the left of (and not including) the `rewrite` position, which represents the schema of the intermediate result (**IR_copy**). The algorithm then removes the original rule, and moves recursively on to the second rule, which contains the remaining body predicates that need to be searched (and possibly split). The recursion terminates at the rightmost predicate position.

3.4 Summary

The delta and localization stages are program rewrites necessary to make materialized (no event predicate) and distributed rules executable. These rewrites are expressed compactly in Overlog (around 12 rules each), and avoid complex C++ code in the *Planner* stage implementation. The original P2 code that performed these tasks consisted of a few hundred lines of code spread throughout the system implementation; making it hard to evolve.

The Evita Raced metacompilation framework allows Overlog compilation tasks to be written in Overlog and executed in the P2 runtime engine. It provides significant

extensibility via a relatively clean declarative language. As we will see next, many of the tasks of query optimization – dynamic programming, dependency-graph construction and analysis, statistics gathering – appear to be well served by a recursive query language. The notion of metacompilation also leads to a very tight implementation with significant reuse of code needed for runtime processing.

Chapter 4

Declarative Rewrite: Magic-sets

Having described the Evita Raced infrastructure, we now turn to our use of it to specify query optimizations in Overlog. Using Evita Raced, we have implemented three optimization techniques from the literature: the magic-sets rewrite [15, 16], the System R dynamic program [82] and the Cascades branch-and-bound algorithm [36]. We begin in this chapter with the magic-sets rewrite, which aims to efficiently answer predicates pertaining to a small subset of the data values in the database. For example, the `shortestPath` predicate in Figure 2.7 pertains to paths originating from node "localhost:10000." In order to efficiently evaluate this predicate, the magic-sets rewrite pushes predicate constants down into the supporting rules so that a Datalog evaluator *never* derives superfluous facts.

As mentioned in Chapter 2.1.3, Datalog-oriented systems like P2 perform a bottom-up (*forward chaining*) evaluation on each rule, starting with known facts (tuples), and recursively deriving new facts through rule deductions. The advantage of this strategy is that the evaluation is data driven (from known facts to possible deductions) and will not enter infinite loops for some statically verifiable *safe* programs. In contrast, a top-down (*backward chaining*) evaluation (e.g., in the Prolog language), starts with the "query" predicates (i.e., `shortestPath` in Figure 2.7) as the top-level goals, and recursively identifies rules whose head predicates unify with needed goals, replacing them with the subgoal predicates in the rule body, until all subgoals are satisfied by known facts or rejected when no further recursion is possible. The advantage of a top-down evaluation strategy is that it avoids resolving goals that are not needed by the posed queries (e.g., paths not originating from "localhost:10000").

For a given Datalog program, the magic-sets rewrite adds extra rules and predicates that prune results, known to be superfluous, from a bottom-up evaluation. The primary data structure used by this rewrite technique is a rule/goal graph, which we review in Chapter 4.1, along with the magic-sets algorithm. Using the rule/goal graph representation of an Overlog program, in Chapter 4.2 we express the magic-sets rewrite in 44 Overlog rules. We divide these rules into two logical groups. The first is presented in Chapter 4.2.1, which constructs the rule/goal graph via a transitive closure on the Metacompiler Catalog. Our second group of rules, presented in Chap-

```
link(``node1'', ``node2'', 1).
link(``node1'', ``node3'', 1).
link(``node2'', ``node1'', 1).
...
r1 path(@X, Y, P, C) :-
   link(@X, Y, C), P := f_cons(X, Y).

r2 path(@X, Z, P, C) :-
   link(@X, Y, C1), path(@Y, Z, P2, C2),
   f_contains(X, P2) == false,
   P := f_cons(X, P2), C := C1 + C2.

query path(``node1'', Y, P, C).
```

Figure 4.1: The path-only rules copied from Figure 2.7.

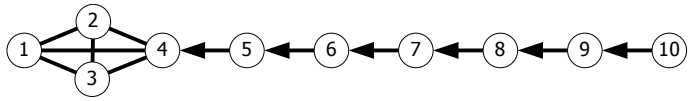


Figure 4.2: Experimental topology.

ter 4.2.2, also performs a transitive closure, but this time on the rule/goal graph itself, to obtain the rewritten rules that include the predicates used to filter tuples that are not relevant to the final answer.

4.1 Magic-sets in a Nutshell

The magic-sets technique rewrites logical rules so that bottom-up evaluation over the rewritten rules has all the advantages of a top-down and a bottom-up evaluation strategy. We give some intuition here by reviewing the advantages of magic-sets using the path program shown in Figure 4.1. For the purpose of this discussion, lets assume we execute these rules locally with the initial set of `link` fact tuples forming the topology shown in Figure 4.2. The abbreviated list of facts shown at the beginning of Figure 4.1 populate the `link` relation with our basic topology information. Our goal is to find all paths that start at "node1."

A straightforward bottom-up evaluation of this program applies the `link` tuples to rule `r1`, creating the initial set of `path` tuples. Rule `r2` performs a transitive closure over the `link` and `path` relations, while any `path` tuples matching "node1" in the first field are returned in the programmer's query. Clearly this bottom-up evaluation strategy examines some `path` tuples that do not contribute to the query answer; for example, paths that originate from nodes $5-10$ [1]. In contrast, a top-down evaluation begins by unifying the query predicate with the head predicate of rules `r1` and `r2`.

[1]Paths that originate from nodes $2-4$ are still relevant since they can be included in paths originating from "node1."

```
link(``node1'', ``node2'', 1).
link(``node1'', ``node3'', 1).
link(``node2'', ``node1'', 1).
...
magic_path(``node1'').

r1_case5 path(@X, Y, P, C) :-
         magic_path(@X),
         link(@X, Y, C), P := f_cons(X, Y).

r2_case2 sup_r2_1(@X, Y, C1) :-
         magic_path(@X),
         link(@X, Y, C1).

r2_case3 magic_path(@Y) :-
         sup_r2_1(@X, Y, C1).

r2_case4 sup_r2_2(@X, Y, Z, C1, P2, C2) :-
         sup_r2_1(@X, Y, C1),
         path(@Y, Z, P2, C2).

r2_case5 path(@X, Z, P, C) :-
         sup_r2_2(@X, Y, Z, C1, P2, C2),
         f_contains(X, P2) == false,
         P := f_cons(X, P2), C := C1 + C2.

query path(``node1'', Y, P, C).
```

Figure 4.3: A magic-sets rewrite of the rules in Figure 4.1.

This `path` predicate unification binds the $@X$ attribute to "node1" in both rules, which is then carried over to the predicates in the rule body.

The magic-sets rewrite is an optimization that can achieve the same efficiency found in the top-down evaluation, using a bottom-up evaluator. Since it is still bottom-up, we retain all the benefits of seminaïve evaluation: set-oriented evaluations, unique minimal model and stratification. Magic-sets does this by adding extra selection predicates to the rules of a program that emulate the goal-oriented execution of a top-down evaluation (sometimes called *sideways information passing* or SIP). Conceptually, given a rule of the form H :- $G_1, G_2, ..., G_k$, where H is the head predicate and $G_{1,...,k}$ are the subgoals, the magic-sets rewrite intersperses selection predicates $s_{1,...,k}$ to generate the rule form H :- $s_1, G_1, s_2, G_2, ..., s_k, G_k$. Facts for these selection predicates are generated according to attribute bindings in the user's query or from other rule predicates in the program, to constant values.

Figure 4.3 shows the rewritten rules from the path program in Figure 4.1. The program contains some new predicates prefixed with `magic_` and `sup_` that are included in the rule body with the `link` and `path` predicates. Ullman [94] refers to these new predicates as magic predicates (i.e., `magic_path`) and supplementary predicates (i.e., `sup_r2_1`, `sup_r2_2`). Magic predicates maintain bindings relevant to query predicates

(i.e., `path`), while supplementary predicates pass bindings along rule bodies, ensuring that no extraneous deductions are made along the way.

We now describe the rewritten rules, which are named by 1) the original rule name, and 2) a "case" number that will be described in Chapter 4.2.2. [2] We start with rule `r1` in Figure 4.1, again assuming we are running locally with all facts in the `link` relation. As it stands, rule `r1` will generate `path` tuples using any of the `link` tuples, regardless of whether they contribute to answering the final query. To avoid extraneous deductions, we add the `magic_path` predicate to the body of this rule, giving us rule `r1_case5` in Figure 4.3.

The rewrite of rule `r2` appears to be quite a bit more complicated, expanding out to four separate rules. We describe the purpose of each rule on a case-by-case basis. Rule `r2_case2` adds a rule that fills the `sup_r2_1` relation with tuples produced by "joining" `magic_path` and `link` relations. The outcome of which is no different than rule `r1_case5` in our previous discussion; excluding the extra path information. The interesting bit here is that, in rule `r2_case4`, the `sup_r2_1` predicate is "joined" with the `path` predicate. This effectively uses the `magic_path` predicate to prune superfluous tuples from `link` before "joining" with the `path` relation.

This brings us to the more interesting case 3 w.r.t. rule `r2`. Here we are feeding `sup_r2_1` tuples into the `magic_path` relation. At a high-level, this rule updates the `magic_path` table with tuples that satisfy the constraints imposed by the current `magic_path` table instance and includes the new (path) information from the `link` predicate. Observe that rule `r2_case3` feeds `magic_path` values from its Y variable, which represents the intermediate hop in rule `r2`, and therefore must be part of the final answer. In this example, rule `r2_case3` is responsible for adding each node in the clique (i.e., nodes 2, 3, and 4 in Figure 4.2) to the `magic_path` relation because there is a path from "node1" to it.

The remaining cases simply stitch things up using the remaining terms in rule `r2`. In case 4, we combine `sup_r2_1` with the `path` predicate to obtain the `sup_r2_2`, which is then used to finish off the rule in case 5 (rule `r2_case5`). The reader may be confused by the need for `sup_r2_2`. Why not simply create the following rule?

```
r2_case? path(@X, Z, P, C) :-
         sup_r2_1(@X, Y, C1),
         path(@Y, Z, P2, C2).
         f_contains(X, P2) == false,
         P := f_cons(X, P2), C := C1 + C2.
```

Indeed, this rule is correct and it does not generate paths that are not relevant to the final answer. Nevertheless, we introduce the `sup_r2_2` predicate (case 4), in general, since we do not know if this is the last occurrence of a magic predicate (i.e., `path`) in rule `r2`. The occurence of a magic predicate p in rule r at position j triggers cases 2 and 3, which generate rules in the following form.

[2] Case 1 refers to the creation of the single magic predicate `magic_path` fact, which contains the bound constants from the query predicate.

- case 2: $sup_j^r(...)$:-$sup_{i-1}^r(...), G_i, ..., G_{j-1}$
- case 3: $magic_p(...)$:-$sup_j^r(...)$

The subgoals $G_{i...j-1}$ refer to EDB predicates appearing in the body of rule r at the respective positions. Returning to our example, case 4 anticipates the need for generating a `sup_r2_X` predicate, which will use `sup_r2_2` and all subseqent EDB predicates to generate case 2. Furthermore, case 3 requires `sup_r2_X` to update the magic (IDB) predicate appearing in rule `r2` at position `X`. In keeping with the current numbering scheme, we note that `sup_r2_0 :- magic_path`.

Before presenting the declarative rules that implement this rewrite technique, we must review the concept of adornments and the rule/goal graph representation for a collection of Datalog (Overlog) rules. These data structures form the basis of the transitive closure algorithm performed by our magic-sets rewrite. The discussion leading up to Chapter 4.2 follows from Chapter 13 of Ullman's textbook [94], which provides the most through coverage on the subject to date.

4.1.1 Adornments

Consider again the path program in Figure 4.1. The query predicate `path(''node1'', Y, P, C)` asks for all paths that originate from "node1". An *adornment* is a binding pattern that contains a string of *b's* (bound) and *f's* (free) of length k, for each k arguments of `path`. In the current context, the *path* query predicate matches the $path^{bfff}$ *adornment* since the first argument is bound to a constant and the last three variables are free to take on any value. Such *goal adornments* are assigned to rule predicates, based on the position of the predicate in the rule and the bindings associated with that rule position.

Rule bindings are assigned by position, according to a left-to-right (SIP) evaluation order. The steps for assigning *rule adornments* are as follows.

1. A variable appearing in a bound argument of the rule head is bound before processing any subgoals i.e., $path^{bfff}$ binds @X in the `path` head of rule `r2`.
2. A variable is bound after processing subgoal G_i if it was bound before processing G_i or if it appears anywhere in G_i i.e., the `link` subgoal binds the @Y variable in the `path` subgoal of rule `r2` (variables Z, $P2$, and $C2$ in `path` remain free).

The format of a rule adornment differs from that of a predicate. It follows the form $[B_1, \cdots, B_m|F_1, \cdots, F_n]$, which contains two sublists of variables separated by a bar. The variables to the left of the bar (i.e., $B_1, \cdots, B_m$) represent bound variables, while those to the right (i.e., $F_1, \cdots, F_n$) are free.

A given rule contains a number of these binding patterns, one for each subgoal position. That is, a rule adornment is a binding pattern of a rule at a given rule

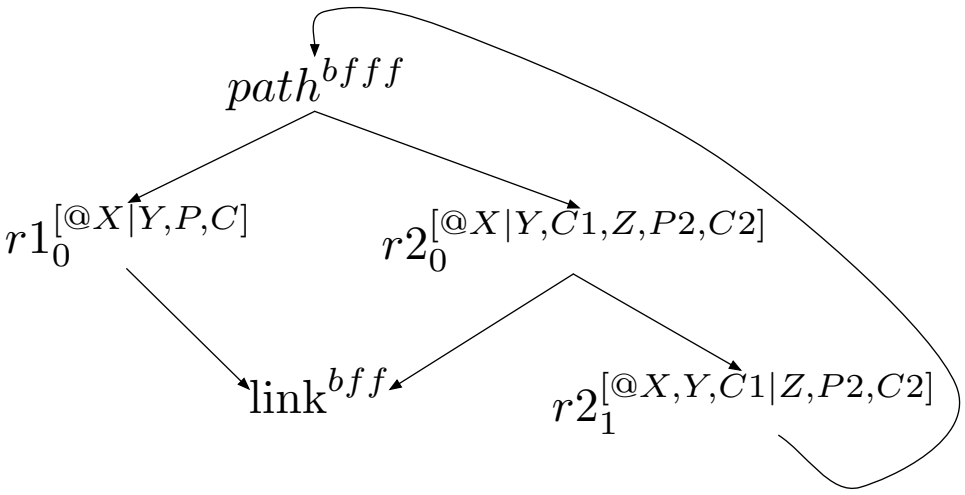


Figure 4.4: Rule/Goal graph of the program in Figure 4.1.

position. The notation that we follow here identifies the rule's position as a subscript and the binding pattern as a superscript. For example, $r1_0^{[@X|Y,P,C]}$ is the adornment for rule `r1` at position 0, which is based on the binding pattern of the $path^{bfff}$ adornment relative to the head predicate schema.

Continuing, $r2_0^{[@X,|Y,Z,C1,P,C]}$ represents the adornment for rule `r2` at the head position 0, again binding the first variable of the head predicate. The $r1_0$ and $r2_0$ rule adornments "feed" the `link` subgoal with its bindings, create the $link^{bff}$ goal adornment. The `link` subgoal adds variables Y and $C1$ to the list of bound variables for rule `r2` at position 1. This yields the $r2_1^{[@X,Y,C1,|Z,P,C]}$ rule adornment, which "feed" bindings into the `path` predicate, creating the $path^{bfff}$ adornment by binding the Y variable in the first argument.

4.1.2 Rule/Goal Graphs

A rule/goal graph is a representation of binding patterns that occur in a collection of Datalog (Overlog) rules. The graph consists of *rule* and *goal* vertices. A goal vertex consists of a predicate with an adornment (e.g., $path^{bfff}$) and similarly, a rule vertex represents the adornment of the rule in a particular position (i.e., $r1_0^{[@X|Y,P,C]}$).

Figure 4.4 illustrates the *rule/goal* graph for our Figure 4.1 example. To construct this graph, we start at the query predicate, and create a goal vertex in the graph with its proper adornment. For every rule with that goal predicate as its head, we create a rule adornment relative to position 0. For rule `r1` this is $r1_0^{[@X|Y,P,C]}$ and for rule `r2` we have $r2_0^{[@X,|Y,Z,C1,P,C]}$. A rule vertex feeds bindings to the subgoal just beyond its position. Both rules in position 0 bind the `link` predicate variable X. In the case of rule `r2`, position 1 receives the bindings of its parent rule and the bindings from the `link` subgoal, giving us the $r1_1^{[@X,Y,C1,|Z,P,C]}$ rule vertex.

At this point in rule `r2` we have reached the position prior to the `path` predicate. We create the appropriate $path^{bfff}$ adornment, which matches up with our original `path` goal node. Since we have no further rule binding steps beyond the path predicate, the process halts. Our declarative rules initiate the rewrite process by performing these steps recursively over the Metacompiler Catalog.

4.2 Declarative Magic-sets

Using Evita Raced, we expressed the magic-sets rewrite stage in Overlog. The first step in this rewrite constructs the rule/goal graph, captured as relational data. It uses this graph data to check for a *unique binding property* with respect to the adornment of the query predicate. This property is met when the query predicate q_p, expressed against predicate p, contains a unique "binding pattern" throughout the rule/goal graph. The query predicate q_p provides the first binding pattern (the root of the rule/goal graph), while rules that mention p provide further bindings based on sideways-information-passing (SIP).

We check for the unique binding property in the first phase of our rewrite, while constructing the rule/goal graph via a transitive closure on the Metacompiler Catalog. If this property is violated at any point then the rewrite terminates early, without changing any rules. The Metacompiler Catalog already provides some of the rule/goal graph information, specifically goal (head predicate) and subgoal (body terms) rule dependencies. The remaining information that we need to collect is the adornments for rule/goal "vertices." Once this information is secured, we can move to the actual rewrite rules described in Chapter 4.2.2, where magic and supplementary relations are created according to the five cases previously discussed.

4.2.1 Rule/Goal Graph Construction

The algorithm for constructing a rule/goal graph begins at the query predicate, and follows with the rules that mention the query predicate in the head. We assume the unique binding property holds in the beginning, and detect if it does not along the way. Given a query predicate q_p, we create a *magic predicate*, denoted as m_p, with a corresponding adornment. A set of *supplementary predicates*, denoted as sup_i (i being the rule position), are also created as we recursively walk the rules in a left-to-right (SIP) order.

The abbreviated rule in Figure 4.5 creates an adornment for the query predicate and adds that fact to the `magicPred` relation. A query predicate is identified in P2 by a rule containing a single goal ($Goals == 1$). The sole predicate in this rule has a schema ($Schema$) that contains some number of (binding) constants and (free) variables. The function `f_adornment` takes such a schema object as its argument and returns a string representing an adornment signature (the binding pattern).

```
/*Create an adornment for the query predicate and add a fact to the magicPred
table referencing this adornment. */
ms1 magicPred(@A, Pid, Name, Sig) :-
    magic::programEvent(@A, Pid, ...),
    sys::rule(@A, Rid, Pid, ..., Goals),
    sys::predicate(@A, _, Rid, ..., Schema),
    Goals == 1,
    Sig := f_adornment(Schema).
```

Figure 4.5: Construction of the query adornment and corresponding magic predicate.

Rule `ms1` creates the top-level goal node that represents the root of the rule/goal graph. The group of rules in Figure 4.6 deal with creating adornments for rule positions in the target program. We store the rule adornment in a `sup` relation since this information will be used to create supplementary predicates in Chapter 4.2.2. A `sup` tuple contains the following attributes in order:

- A reference to the target program and rule identifiers.
- A position within that target rule.
- A name for the supplementary predicate.
- A rule adornment, as a schema object containing all constants and variables up to that rule position.
- A new identifier that will be used (in Chapter 4.2.2) to create a new rule that supplies facts to the supplementary relation (e.g., rule cases 2 and 4 in Figure 4.3).

We now describe the details of each rule in Figure 4.6. Rule `ms2` initiates the first `sup` deduction. It joins `magicPred` with the `rule` and `predicate` relations to obtain those rules that reference a magic predicate in the head. This result will represent the supplementary predicate in position 0. The adornment for this rule position is obtained by projecting the head predicate schema onto the magic predicate adornment. The function `f_project` takes care of the step-by-step details of combining the head predicate schema and the signature of the magic predicate adornment, and then returning a new schema that contains only the bound head variables (according to the adornment). For example, if the head predicate schema is $[@X, Y, P, C]$ and the adornment is $bfff$ then the `f_project` will return $[@X]$ as the new schema. The new schema is used by the current (position 0) supplementary predicate. [3]

The rules that receive a `sup` tuple are consider next by creating further `sup` tuples for each subgoal position. In Overlog, only table predicates and "assignment" statements create new bindings. As a result, `sup` tuples are only generated for rule

[3]The supplementary predicate at position 0 is a symbolic reference to the magic predicate.

```
/*Initialize sup position 0 for rules that reference a magic predicate in the head. */
ms2 sup(@A, Pid, Rid, Pos, SupName, Schema, f_idgen()) :-
    magicPred(@A, Pid, Name, Sig),
    sys::rule(@A, Rid, Pid, RName, HeadPid, ...),
    sys::predicate(@A, HeadPid, Rid, _, Name, ..., FSchema, ...),
    Schema := f_project(Sig, FSchema),
    SupName := "sup_" + RName + 0,
    Pos := 0.

/*Create supplementary predicate for a given subgoal. */
ms3 sup(@A, Pid, Rid, Pos, SupName, NewSchema, f_idgen()) :-
    supNext(@A, Pid, Rid, Pos, Schema),
    sys::rule(@A, Rid, Pid, RName, ...),
    sys::predicate(@A, Fid, Rid, ..., FSchema, Pos, ...),
    SupName := "sup_" + RName + "_" + Pos,
    NewSchema := f_merge(Schema, FSchema).

/*Create supplementary predicate for a given assignment. */
ms4 sup(@A, Pid, Rid, Pos, SupName, NewSchema, f_idgen()) :-
    supNext(@A, Pid, Rid, Pos, Schema),
    sys::rule(@A, Rid, Pid, RName, ...),
    sys::assign(@A, Aid, Rid, Var, _, Pos),
    SupName := "sup_" + RName + "_" + Pos,
    NewSchema := f_assignschema(Schema, Var).

/*Move the rule position forward when update occurs to sup. */
ms5 supNext(@A, Pid, Rid, Pos+1, Schema) :-
    sup(@A, Pid, Rid, Pos, Name, Schema, Tid).

/*Move supNext forward for selection predicates. */
ms6 supNext(@A, Pid, Rid, Pos+1, Schema) :-
    supNext(@A, Pid, Rid, Pos, Schema),
    sys::rule(@A, Rid, Pid, ..., Goals),
    sys::select(@A, Sid, Rid, _, Pos, _),
    Pos < Goals.
```

Figure 4.6: Rules for supplementary relational predicates.

```
/*We've encountered a magic predicate in the body of a rule. Compute its adornment based on
current bound variables. */
ms7 magicPred(@A, Pid, FName, Sig) :-
    supNext(@A, Pid, Rid, Pos, Schema),
    sys::rule(@A, Rid, Pid, RName, ...),
    sys::predicate(@A, Fid, Rid, _, FName, ..., FSchema, Pos, ...),
    magicPred(@A, Pid, FName, Sig),
    Sig := f_adornment(Schema, FSchema).
```

Figure 4.7: Encountering a magic predicate during subgoal traversal.

positions relevant to such terms: relational predicates (rule `ms3`) and assignment statements (rule `ms4`). The `f_merge` and `f_assignschema` functions are used to update the schema object with the bindings of the current term position. A series of `supNext` tuples is created for each rule position to considered. The `supNext` relation is generated by rule `ms5` for predicate and assignment terms, and rule `ms6` for selection predicate terms, which add no new bindings to the previous schema.

The previous group of rules ignored the special case of discovering a magic predicate at rule positions referenced by some `supNext` tuples. In order to verify that the unique binding property holds, we must compute the adornment for each magic predicate appearance in the rule body. Figure 4.7 contains the single rule that generates an adornment for a subgoal that references a magic predicate. If the adornment is different than the previous, then multiple rows will exist in the `magicPred` relation, signaling the presence of multiple magic predicate binding patterns. A simple count query (Figure 4.8: count `ms12` and check `ms13`) is used to detect violations of the unique binding property.

The last group of rules detect when the rule/goal graph construction "phase" has completed, and on completion, checks for the unique binding property. In Figure 4.8, rules `ms9` and `ms10` together count the number of rules that have completed the rule/goal graph construction phase. Rule `ms11` counts the total number of rules in a given program and rule `ms12` counts the number of adornments for a given `magicPred`. Finally, rule `ms13` signals the completion of the current phase by deriving a `commitMagicPred` tuple if all rules have completed and the magic predicate has a single adornment. We note here that the counts for `programRuleCount` and `rulesComplete` would not be needed if P2 had support stratified Datalog. The magic predicate adornment count is needed before moving to the next phase, but it also marks a stratification boundary. To prevent a premature `commitMagicPred` deduction in rule `ms13`, we ensure the counts in `programRuleCount` and `rulesComplete` are equal.

4.2.2 Rewrite Phase

At this point, the adornment information for the magic predicate and rule positions have been populated in the `magicPred` and `sup` relations, and we can now begin with

```
/*Indicate when a rule has been fully explored. */
ms9 ruleComplete(@A, Pid, Rid) :-
    supNext(@A, Pid, Rid, Pos, _),
    sys::rule(@A, Rid, Pid, ..., Goals),
    Pos == Goals.

/*Count the number of completed rules. */
ms10 rulesComplete(@A, Pid, a_count<Rid>) :-
     ruleComplete(@A, Pid, Rid).

/*Count the number of rules in a program. */
ms11 programRuleCount(@A, Pid, a_count<Rid>) :-
     programEvent(@A, Pid, ...),
     sys::rule(@A, Rid, Pid, ...).

/*Count the number of adornments for a given magic predicate. */
ms12 countAdornments(@A, Pid, Name, a_count<Sig>) :-
     magicPred(@A, Pid, Name, Sig).

/*Commit a magic predicate iff it has a unique adornment. */
ms13 commitMagicPred(@A, Pid, Name, Sig, f_idgen()) :-
     programRuleCount(@A, Pid, RuleCount),
     rulesComplete(@A, Pid, RuleCount),
     countAdornments(@A, Pid, Name, Count),
     magicPred(@A, Pid, Name, Sig),
     Count == 1.
```

Figure 4.8: Detect completion of rule/goal traversal and check for unique binding property.

```
/*Create a rewriteRule tuple that contains identifiers for a new ruleand a corresponding head
predicate. */
ms14 rewriteRule(@A, Pid, Rid, f_idgen(), f_idgen(), MagicName, Sig) :-
       commitMagicPred(@A, Pid, MagicName, Sig, Tid),
       sys::rule(@A, Rid, Pid, Rid, HeadID, ...),
       sys::predicate(@A, HeadID, Rid, _, PredName, ...),
       f_isMagicPredName(PredName, MagicPredName) == true.
```

Figure 4.9: Signal the rewrite of the top level rule containing the given magic predicate.

the actual rewrite phase. We now further describe the cases mentioned in Figure 4.3 in a general fashion. Consider the following rule with k subgoals and a query predicate p

p :- $G_1, \cdots, p, \cdots, G_k$.

The head and the i^{th} subgoal both reference predicate p. Our magic-sets rules will rewrite the above rule into the following rule cases.

1. **case 1:** $m_p(\cdots)$.
2. **case 2:** sup_{i-1} :- $m_p,\ G_1,\ \cdots,\ G_{i-1}$.
3. **case 3:** m_p :- sup_{i-1}.
4. **case 4:** sup_i :- $sup_{i-1},\ p$.
5. **case 5:** p :- $sup_i,\ G_{i+1}, \cdots, G_k$.

These rule cases reference the original goals $G_\#$ and head predicate p, along with new magic (m_p) and supplementary ($sup_\#$) predicates.

We now give a high level description of each case in order. The first is simply a fact on the magic predicate m_p, containing the constants mentioned in the query predicate p. The second case creates a rule body containing the magic predicate m_p and the first $i-1$ subgoals (prior to the p predicate position). The rule head for this second case references the supplementary relation sup_{i-1}. The third case has the supplementary predicate sup_{i-1} feeding the magic predicate m_p values, taken from the SIP sup_{i-1} bindings. The fourth case joins sup_{i-1} with predicate p (the i^{th} subgoal) to supply the values for the sup_i head predicate. Finally, in the fifth case, we complete the rule by joining sup_i with the remaining subgoals, and projecting that result onto the original head predicate p. We now describe these steps declaratively.

```
/*The event predicate for the new rule is the magic predicate, which through
sideways information passing will trigger the rule's execution. */
ms15 sys::predicate(@A, f_idgen(), NewRid, false, MagicNameName, Tid,
                    ''DELTA'', MagicSchema, 1) :-
     rewriteRule(@A, Pid, Rid, NewRid, NewHead, MagicName, MagicSig),
     sup(@A, Pid, Rid, 0, Name, Schema, Tid),
     MagicSchema := f_project(MagicSig, Schema).

/*Initiate an iterator for the new magic predicate rewrite along a given rule.
The iteration begins at the goal predicate immediately following the event
predicate. */
ms16 rewriteIter(@A, Pid, Rid, NewRid, NewHeadFid, 1, 2) :-
     rewriteRule(@A, Pid, Rid, NewRid, NewHeadFid, _, _).
```

Figure 4.10: Rule for initiating an iteration over the top level rule that is to be rewritten.

Initialization

Figure 4.9 contains rule `ms14`, which initializes the rewrite phase from the magic predicate reference contained in the `commitMagicPred` tuple [4]. The rule derives a `rewriteRule` tuple for each rule with a head predicate that matches an existing magic predicate. The schema of `rewriteRule` contains attributes that hold new identifiers for a new rule, and corresponding head predicate, that will handle **case 3** and **case 5**, depending on a condition we defer for now.

The `rewriteRule` predicate is used in Figure 4.10 to create the magic predicate m_p in the event position (one) and to initiate a `rewriteIter` tuple. Rule `ms15` concurrently handles the magic predicate for **cases 2 and 5**, using the rule/goal graph information for `sup` position 0. The next step is to walk down the list of subgoals in the original rule body and copy each subgoal G_i that does not reference a magic predicate to the new rule. Rule `ms16` takes care of invoking this fact through a `rewriteIter` tuple with the following information.

1. Location attribute.
2. Program identifier.
3. The original rule identifier.
4. A new rule identifier.
5. An identifier for the new rule's head predicate.
6. The subgoal position relative to the original rule.
7. A position of the subgoal in the new rule.

```
/*If goal node G_i is not a magic predicate then shift position to NewPos
in the new rule NewRid. */
ms17 sys::predicate(@A, PredID, NewRid, NotIn, Name, Tid, ECA, Schema,
                    NewPos) :-
     rewriteIter(@A, Pid, Rid, NewRid, NewHeadFid, RulePos, NewPos),
     sys::predicate(@A, PredID, Rid, NotIn, Name, Tid, ECA, Schema,
                    RulePos),
     notin magicPred(@A, Pid, Name, Sig).

/*Point assignment to the new rule (NewRid) in the new position (NewPos). */
ms18 sys::assign(@A, Aid, NewRid, Var, Value, NewPos) :-
     rewriteIter(@A, Pid, Rid, NewRid, NewHeadFid, RulePos, NewPos),
     sys::assign(@A, Aid, Rid, Var, Value, RulePos).

/*Point selection predicate to the new rule (NewRid) in the new position (NewPos). */
ms19 sys::select(@A, Sid, NewRid, Bool, NewPos) :-
     rewriteIter(@A, Pid, Rid, NewRid, NewHeadFid, RulePos, NewPos),
     sys::select(@A, Sid, Rid, Bool, RulePos).
```

Figure 4.11: Rule's for moving subgoals in the top level rule the new rule undergoing the rewrite.

The primary purpose of the `rewriteIter` is to reference the subgoals of the original rule leading up to a predicate that references a magic predicate. These prior subgoals need to be copied to the new rule. This is handled by the rules in Figure 4.11. Rule `ms17` copies the predicate at position *RulePos* (starting at position 1) in the original rule to position *NewPos* (starting at position 2, just after the "magic" event predicate) in the new rule. Rules `ms18` and `ms19` simply copy EDB subgoals — including assignment and selection predicates — in the old rule to the new rule.

Figure 4.12 contains two rules that will either move the positions referenced in the current `rewriteIter` forward, or deduce a new `break` tuple. These two conditions are based on the current subgoal at position *RulePos*, and whether it references a magic predicate. If not, then rule `ms20` advances the `rewriteIter` positions (both *RulePos* and *NewPos*) by one. Otherwise, rule `ms21` derives a `break` tuple that contains the new identifiers associated the new rule.

Are we there yet?

We now need to consider whether we have completed the rewrite for a given target rule, or not. This decision is based on the rule position referenced in the `break` tuple. If at that position lies a magic subgoal, then we must finalize the rule for **case 2**, and create the rules for **case 3** and **case 4**. If it occurs after the last subgoal position then we simply finalize the rule in **case 5**, which completes the rewrite for the given target

[4]The planner uses tuples in `commitMagicPred` to create the necessary magic predicate facts in **case 1**.

```
/*Continue the rewrite iter if the current goal node Pid is not a magic predicate. */
ms20 rewriteIter(@A, Pid, Rid, NewRid, HeadFid, RulePos+1, NewPos+1) :-
     rewriteIter(@A, Pid, Rid, NewRid, HeadFid, RulePos, NewPos),
     sys::predicate(@A, Pid, Rid, NotIn, Name, Tid, ECA, Schema,
                    RulePos),
     notin magicPred(@A, Pid, Name, Sig).

/*The current goal node Pid is a magic predicate. Indicate where the break
occurs (RulePos) within the subgoals of the given rule Rid. */
ms21 break(@A, Pid, Rid, NewRid, NewHeadID, RulePos, NewPos) :-
     rewriteIter(@A, Pid, Rid, NewRid, NewHeadID, RulePos, NewPos),
     sys::predicate(@A, Pid, Rid, _, Name, _, _, Schema, RulePos),
     magicPred(@A, Pid, Name, Sig).
```

Figure 4.12: Given a particular subgoal G_i, these rules determine if the iteration should continue to the next subgoal or if a **break** tuple should be deduced because G_i represents a magic predicate.

rule. We consider here, the case when we arrive at a magic subgoal, and conclude this section with a description of the final case.

Not yet

Recall that the rules in Figure 4.11 copy subgoals over to the new **case 2** (or perhaps the **case 5**) rule, as these subgoals are referenced by `rewriteIter` tuples. Furthermore, rule `ms15` in Figure 4.10 already created the m_p predicate (set to the magic predicate) in the event position of our new **case 2** rule. Therefore, all that remains is for us to deal with the final head predicate sup_{i-1}, in this case.

Figure 4.13 contains the rules that finalize **case 2**. Rule `ms22` generates a `sup_case2` tuple if the *RulePos* does not exceed the number of terms in the rule body. In rule `ms23`, we reference the supplementary predicate sup_{i-1} in the head of the rule. And finally in rule `ms24`, we commit this rule information to the `rule` relation, indicating the relevant identifiers and the number of terms ($NewPos$) in it.

Figure 4.14 contains the rules that handle **case 3**. Similar to the previous rules, we initiate this rewrite in rule `ms25` if the *RulePos* refers to an actual subgoal, which itself is implicitly referencing a magic predicate. Rule `ms26` creates the magic predicate m_p head for our **case 3** rule, while rule `ms27` creates a reference to the sup_{i-1} supplementary predicate in the event position. We commit our **case 3** rule in rule `ms28`, which indicates that the new rule contains exactly two terms.

Figure 4.15 contains the rules that deal with **case 4**. The familiar rule `ms29`, derives a `sup_case4` tuple, which contains new identifiers for the new rule and its head predicate. The sup_i predicate information is obtained from the `sup` relations at the *RulePos* position. This supplementary predicate will be the head predicate in the **case 4** rule, and it is created by rule `ms30`. We then come to rule `ms31`, which

```
ms22 sup_case2(@A, Pid, Rid, NewRid, NewHeadID, RulePos, NewPos) :-
     break(@A, Pid, Rid, NewRid, NewHeadID, RulePos, NewPos),
     sys::rule(@A, Rid, Pid, Rid, ..., Terms),
     RulePos < Terms.

 /*Write predicate sup_{i-1} to predicate relation in head position 0. */
ms23 sys::predicate(@A, NewHeadFid, NewRid, false, SupName, SupTid,
                        null, Schema, 0) :-
     sup_case2(@A, Pid, Rid, NewRid, NewHeadFid, RulePos, _),
     sup(@A, Pid, Rid, SupPos, SupName, Schema, SupTid),
     SupPos == RulePos - 1.

 /*Commit this rule. */
ms24 sys::rule(@A, NewRid, Pid, RuleName, NewHeadID, null, false,
                NewPos) :-
     sup_case2(@A, Pid, Rid, NewRid, NewHeadID, RulePos, NewPos),
     RName := "SupRule" + Rid + RulePos.
```

Figure 4.13: Finalize **case 2:** sup_{i-1}:-$sup_{i-j}, G_j, G_{j+1}, \cdots, G_{i-1}$.

```
 /*Initiate this rewrite by inferring a sup_case2 tuple with required information. */
ms25 sup_case3(@A, Pid, Rid, f_idgen(), f_idgen(), RulePos) :-
     break(@A, Pid, Rid, _, _, RulePos, NewPos),
     sys::rule(@A, Rid, Pid, Rid, ..., Terms),
     RulePos < Terms.

 /*Create m_p magic head predicate in the new rule. */
ms26 sys::predicate(@A, NewHeadID, NewRid, false, MagicPredName, Tid,
                        null, MagicSchema, 0) :-
     sup_case3(@A, Pid, Rid, NewRid, NewHeadID, RulePos),
     sup(@A, Pid, Rid, RulePos, Name, SupSchema, _),
     SupPos == RulePos - 1$,
     commitMagicPred(@A, Pid, Name, Sig, Tid),
     MagicSchema := f_project(Sig, SupSchema).

 /*Create the supplementary predicate sup_{i-1} (i == RulePos) in the new rule. */
ms27 sys::predicate(@A, f_idgen(), NewRid, false, Name, Tid,
                        "DELTA", Schema, 1) :-
     sup_case3(@A, Pid, Rid, NewRid, _, RulePos),
     sup(@A, Pid, Rid, SupPos, Name, Schema, Tid),
     SupPos == RulePos - 1.

 /*Commit the new rule with 3 terms. */
ms28 sys::rule(@A, NewRid, Pid, RuleName, NewHeadID, null, false, 2) :-
     sup_case3(@A, Pid, Rid, NewRid, NewHeadID, RulePos),
     RuleName := "MagicPredFill" + Rid + Pos.
```

Figure 4.14: Create the rule for **case 3**: m_p :- sup_{i-1}.

```
/*Initiate this rewrite by inferring a sup_case2 tuple with required information. */
ms29 sup_case4(@A, Pid, Rid, f_idgen(), f_idgen(), RulePos) :-
     break(@A, Pid, Rid, _, _, RulePos, NewPos),
     sys::rule(@A, Rid, Pid, Rid, \ldots, Terms),
     RulePos < Terms.

/*Create sup_i (i == RulePos) head predicate in the new rule. */
ms30 sys::predicate(@A, NewHeadID, NewRid, false, Name, Tid, null,
                    Schema, 0) :-
     sup_case4(@A, Pid, Rid, NewRid, NewHeadID, RulePos),
     sup(@A, Pid, Rid, RulePos, Name, Schema, Tid),

/*Create the supplementary predicate sup_{i-1} (i == RulePos) in the new rule. */
ms31 sys::predicate(@A, f_idgen(), NewRid, false, Name, Tid, "DELTA",
                    Schema, 1) :-
     sup_case4(@A, Pid, Rid, NewRid, _, RulePos),
     sup(@A, Pid, Rid, SupPos, Name, Schema, Tid),
     SupPos == RulePos - 1.

/*Copy target rule subgoal G_i, which has a magic predicate m_p, to the new rule. */
ms32 sys::predicate(@A, f_idgen(), NewRid, false, Name, Tid,
                    "PROBE", Schema, 2) :-
     sup_case4(@A, Pid, Rid, NewRid, _, RulePos),
     sys::predicate(@A, Pid, Rid, _, Name, Tid, _, Schema, RulePos).

/*Commit the new rule with 3 terms. */
ms33 sys::rule(@A, NewRid, Pid, RuleName, NewHeadID, null, false, 3) :-
     sup_case4(@A, Pid, Rid, NewRid, NewHeadID, RulePos),
     RuleName := "MagicPredFill" + Rid + Pos.
```

Figure 4.15: Create the rule for **case 4**: sup_i :- sup_{i-1}, G_i.

```
/*Restart the rule rewrite process. The restart tuple contains identifiers
for the new rule identifier and its corresponding head predicate. */
ms34 restart(@A, Pid, Rid, f_idgen(), f_idgen(), RulePos) :-
     break(@A, Pid, Rid, NewRid, HeadFid, RulePos, NewPos).
     sys::rule(@A, Rid, Pid, Rid, ..., Terms),
     RulePos < Terms.

/*Create the event predicate for the rule in the next iteration that
references supplementary predicate sup_i. */
ms35 sys::predicate(@A, f_idgen(), NewRid, false, Name, Tid, "DELTA",
                    Schema, 1) :-
     restart(@A, Pid, Rid, NewRid, HeadFid, RulePos),
     sup(@A, Pid, Rid, RulePos, Name, Schema, Tid).

/*Restart iterator by deducing a new rewriteIter tuple containing
the new identifiers (rule and head predicate) and new positions. */
ms36 rewriteIter(@A, Pid, Rid, NewRid, HeadFid, RulePos+1, 2) :-
     restart(@A, Pid, Rid, NewRid, HeadFid, RulePos).
```

Figure 4.16: Rules for starting the next iteration after encountering a magic predicate in the top level rule.

creates a reference to supplementary predicate sup_{i-1} in the event position of the body. The subgoal in the original rule at position *RulePos* is copied to the second position by rule `ms32`. Finally, rule `ms33` completes **case 4** by deriving a `rule` tuple with the appropriate information (i.e., 3 terms).

The rules in Figure 4.16 restart the rewrite traversal over the target rule. The `break` tuple contains the position of the goal node that represents the magic predicate. Rule `ms34` derives a `restart` tuple at the position following the magic predicate and creates a new rule and head predicate identifier (for the next **case 2/5** rule). Rule `ms35` adds the sup_i ($i == RulePos$) supplementary predicate to the first position of the new rule for the next iteration. And finally, rule `ms36` generates a new `rewriteIter` tuple with the new position ($NewPos$), starting at two since sup_i is already at the event position 1.

Finally

Figure 4.17 contains the rules that handle **case 5**, which is similar to **case 2**. The difference here is that we have reached the last term in the original rule. Therefore, we need a finalizer rule, whose body already contains the previous supplementary predicate and subsequent subgoals that do not refer to a magic predicate; all copied during the `rewriteIter` traversal using rules `ms17`, `ms18` and `ms19`. The head of this new rule is given the original head predicate p, which we copy in rule `ms38` by referencing the original rule head predicate, through the original rule identifier *Rid*, and predicate at position 0. Rule `ms37` simply initiates these rules when the *RulePos*

```
/*Create the group record that will contain the new rule identifier
and the new head predicate identifier. */
ms37 sup_case5(@A, Pid, Rid, NewRid, NewHeadID, NewPos) :-
     break(@A, Pid, Rid, NewRid, NewHeadID, RulePos, NewPos),
     sys::rule(@A, Rid, Pid, Rid, ..., Terms),
     RulePos == Terms.

/*Copy the old head predicate to the new rule's head predicate. */
ms38 sys::predicate(@A, NewHeadID, NewRid, false, Name, Tid, null,
                    Schema, 0) :-
     sup_case5(@A, Pid, Rid, NewRid, NewHeadID, _),
     sys::predicate(@A, _, Rid, _, Name, Tid, _, Schema, Pos),
     Pos == 0.

/*Commit the new Rule. */
ms39 sys::rule(@A, NewRid, Pid, RuleName, NewHeadID, null, false,
               NewPos) :-
     sup_case5(@A, Pid, Rid, Pos, NewRid, NewHeadID, NewPos),
     RuleName := "SupRuleGroup3" + Rid + Pos.
```

Figure 4.17: Create the rule for **case 5**: h :- sup_i, $G_{i+1}, \cdots, G_k$. Since we have already copied the body predicates to the new rule, we only need copy the head predicate from the old rule to be the head of the new rule.

position is equal to the number of terms in the original rule. And finally, rule `ms39` commits the new rule information.

4.2.3 Termination

The final step is to detect when the rewrite has completed. On completion, we clean up all references to rewritten rules and update the `program` relation to signal the completion. Since this rewrite spans an unknown number of dataflow fixpoints (Chapter 2.3.1), we must detect the termination of this rewrite manually. Our termination rules are shown in Figure 4.18. Rule `ms40` counts the number of rules that need to be rewritten by counting the number of `commitMagicPred` tuples generated for rules that contain such a predicate in the head position. Rule `ms41` counts the number of rules that have been completely rewritten. This occurs when the `rewriteIter` reaches the final rule term. A completion is derived when these two counts are equal in rule `ms42`. Note that the `rewriteCount` is derived in the rule/goal graph construction phase, while `completeCount` is evaluated in the rewrite phase. As a result, the derivations of these two counts are separated by a dataflow fixpoint boundary, which means that the rewrite is complete when these counts are equal for a given program. Rule `ms43` performs some housekeeping on the `rule` relation, and rule `ms44` returns control to the *StageScheduler*.

```
/*Count the number of rules that will be rewritten. */
ms40 rewriteCount(@A, Pid, a_count<*>) :-
     commitMagicPred(@A, Pid, Name, ...),
     sys::rule(@A, Rid, Pid, Rid, HeadFid, ...),
     sys::predicate(@A, HeadFid, Rid, _, Name, ...).

/*Count the number of rules that have been rewritten. A rule
has been fully rewritten when the rewriteIter Pos is at
the last subgoal at position Goals. */
ms41 completeCount(@A, Pid, a_count<*>) :-
     rewriteIter(@A, Pid, Rid, Pos, ...),
     sys::rule(@A, Rid, Pid, Rid, ..., Goals),
     Pos == Goals.

/*Since P2 does not support stratified Datalog we must manually detect
when the rewrite has completed. */
ms42 rewriteComplete(@A, Pid) :-
     rewriteCount(@A, Pid, Count),
     completeCount(@A, Pid, Count).

/*Cleanup all rules that were rewritten. */
ms43 delete sys::rule(@A, Rid, Pid, Rid, ..., Goals) :-
            rewriteIter(@A, Pid, Rid, Pos, ...),
            rewriteComplete(@A, Pid),
            sys::rule(@A, Rid, Pid, Rid, ..., Goals),
            Pos == Goals.

/*Signal the completion of this rewrite. */
ms44 sys::program(@A, Pid, Name, Rewrite, "magic-sets", Text, Msg,
                  P2DL, Src) :-
     rewriteComplete(@A, Pid),
     sys::program(@A, Pid, Name, Rewrite, Stage, Text, Msg, P2DL, Src).
```

Figure 4.18: Detect the termination of the magic-sets rewrite. On termination, clean up old rule state and signal the completion of the rewrite.

4.2.4 Magic-sets by example

We briefly summarize the high-level points of our two phases relative to its transformation of the path program (Figure 4.1 to Figure 4.3). The rule/goal graph for this program was presented in Figure 4.4. We now focus on the final rewritten program, which was shown in Figure 4.3.

A transitive closure over the rule/goal graph generates magic and supplementary predicates specific to each "goal" vertex in the `magicPred` table. In the example, a single adornment for the `link` and `path` goals. Since the `path` predicate is referenced by the query predicate, it is given the magic predicate `path_magic`. The `path_magic` predicate is inserted in the 1^{st} position of all rules with the `path` predicate as the rule head. The `path_magic` predicate includes the bound variables (i.e., $@X$) from the `path` head predicate relative to the `path` adornment (signature). In the example, the adornment for `path` is $bfff$, which for both rules yields the `magic_path(@X)` predicate. Also *supplementary* predicates are created for rule positions prior to, and at, `path` predicate subgoals. For example, `sup_r2_1(@X,Y,C1)` is created for "rule" vertex $r_{2,1}$ with the bound variables of the `magic_path` and `link` subgoals.

Also during the second phase, the algorithm maintains the magic predicate relation, which was placed within the rewritten program. Any a priori known bindings about the root goal vertex (e.g., from the user's query) are placed in the magic relation. In the example, the fact `magic_path(''node1'')` is put into the database from the bindings in the `path` query. Also, any edges in the rule/goal graph that start from a rule vertex and end at a goal vertex, with a unique adornment (i.e., upward arrows in the recursive tree that constitutes the graph), are written as rules that generate new magic tuples from new tuples of the rule node's supplementary predicate. In the example, rule `r2_case3` adds more magic facts as more `sup_r2_1` tuples are produced.

4.3 Magic-sets in the Network

We conclude with an analysis of the magic-sets rewrite in a networked setting. What is intuitively happening in Figure 4.3 is that the variable bindings in the query are recursively translated into filtering magic and supplementary predicates. Since the query is only looking for paths from "node1", at first the magic fact in rule `r1_case5` restricts single-hop paths created from links to only those that originate from "node1" Similarly, in what used to be rule `r2`, `link` tuples are filtered according to the magic predicate (in rule `r2_case2`), before being joined with existing `path` tuples to complete the old rule `r2`. The reason rule `r2` was split into the four rules is because the supplementary result `sup_r2_1` is needed for adding extra bindings to the `magic_path` table (in rule `r2_case3`); this is because any variable binding that survives filtering right before the `path` predicate in the body of the old rule `r2` is also an interesting binding for existing or future `path` tuples. If the original program had not been

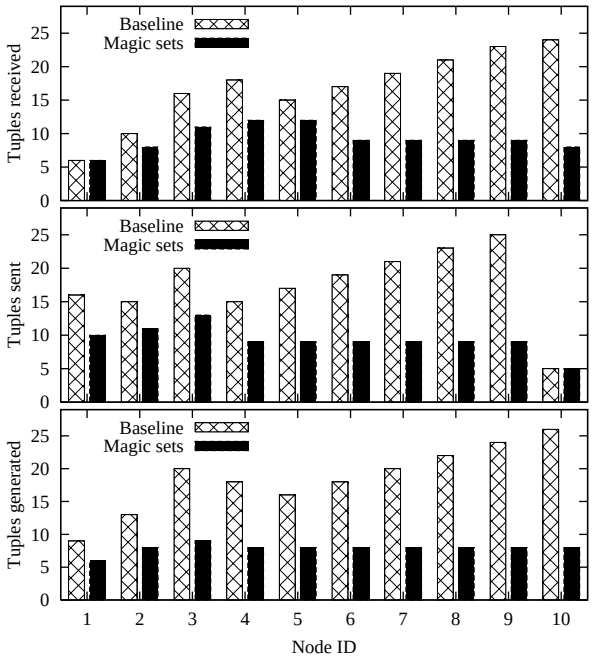


Figure 4.19: For each node (node ID on x axis), number of tuples received (top), sent (middle), and locally generated (bottom) on the y axis.

recursive, then such recursive definitions of magic facts would not appear in the rewritten program.

To understand the effects of this rewrite, we describe two experimental runs of our program, before and after the magic-sets rewrite (both programs were also subjected to the localization rewrite from Chapter 3.3 since they are distributed). The two programs are executed in the simple link topology of Figure 4.2. Nodes are started up one at a time in order of identifier, and the preloaded database (EDB) consists of the links pictured. For each experiment we measure the number of tuples sent and received by each node, as well as any `path` tuples constructed. The latter measure is meant to convey "work" performed by the distributed program even in local computation that does not appear on the network (e.g., local tuple computations, storage, and other dependent actions on those tuples).

Figure 4.19(a) shows the number of tuples that each node receives from the network. The magic-sets rewritten program causes no more tuples to be received than the original, and for most nodes significantly fewer when moving to nodes farther away from the clique. That is because many paths that are generated in the original program with destinations within the clique other than node *node*1 are pruned early on and never transmitted all the way to the far end. Similarly, Figure 4.19(b) shows the number of tuples each node transmits. Again, the magic-rewritten program does a lot better. The two programs have similar tuple transmit/receive overheads for nodes represents the number of tuples a node sends out over the network.

The inclusion of the magic-sets rewrite reduces the number of sends in all but

one case (*node*10). We note here that the edges from *node*10 to *node*4 are directed. As a result, *node*10 is the only node with no incoming links and is therefore never burdened with network traffic other than its own. As a result, its transmit tuple overhead is unaffected, since it already sends out no extraneous paths other than its own path to other nodes. Finally, tuple storage is impacted beneficially by magic sets everywhere (Figure 4.19(c)), since both `path` tuples received from the network, and those generated locally for local consumption are pruned away by the rewrite.

Chapter 5

Declarative Optimization

Previous chapters described the Evita Raced declarative architecture and its reuse of the query executor in a stylized fashion to serve as the engine beneath the query compilation process. This resulted in an *economy of mechanism* [80] not afforded by earlier extensible optimizers (i.e., EXODUS [32], Starburst [75], Volcano [38], OPT++ [52]). In Chapter 4, we presented our first optimization stage; the magic-sets rewrite, which we declaratively expressed as a transitive closure over the rule/goal graph of an Overlog program.

In this chapter we turn our attention to cost-based optimizations, which are commonly based on dynamic programming algorithms. We begin in Chapter 5.1 with a short review of literature on extensible query optimizers, with further details described in the two optimizations we discuss. Chapter 5.2 describes a dynamic programming optimizer stage akin to that of System R. In Chapter 5.3, we present a declarative version of the Cascades branch-and-bound optimizer, which is structured around a dynamic programming algorithm called "memoization." Based on our experience described here, we believe that declarative metacompilation is a clean, architecturally parsimonious way to build the next generation of extensible query optimizers for a wide variety of emerging application domains, where the relevant optimizations are likely to evolve over time.

5.1 Related Work

The pioneering work on extensible query optimizer architectures was done in the EXODUS [32] and Starburst [59, 75] systems, which provided custom rule languages for specifying plan transformations. The EXODUS optimizer generator used a forward-chaining production rule language to iteratively transform existing query plans into new ones. Follow-on work (Volcano [38] and Cascades [36]) exposed more interfaces to make the search in this space of transformations more efficient. Starburst had two rule-based optimization stages. The SQL Query Rewrite stage provided a production rule execution engine, for "rules" that were written imperatively in C; it included a

precedence ordering facility over those rules. The cost-based optimizer in Starburst was more declarative, taking a grammar-based approach to specifying legal plans and subplans.

While all of this work was rule-based and extensible, most of it only exposed individual plan transformations to extensibility; the actual search algorithms or transformation orderings of EXODUS, Volcano, Cascades, and the Starburst cost-based optimizer were confined to procedural code. By contrast, Evita Raced does not embed a search algorithm, instead leaving that open to specification as needed. As we show in Chapter 5.2, both the System R bottom-up strategy and the Cascades top-down strategy naturally fit to a Datalog-based rule language.

Another interesting extensible query optimizer is Opt++ [51], which exploits the object-oriented features of C++ to make an optimizer framework that was easy to customize in a number of ways. A specific goal of Opt++ was to make the search strategy extensible, enabling not only top-down vs. bottom-up state-space enumeration, but also randomized search algorithms. Evita Raced embraces these additional dimensions of extensibility introduced by Opt++, but provides them in a higher-level declarative programming framework.

5.2 System R Optimization

The System R optimizer paper by Selinger, et al. is the canonical textbook framework for database query optimization [82]. The paper laid out for the first time the notion that query optimization can be decomposed into two basic parts: query plan cost estimation and plan enumeration. While this algorithm is traditionally implemented inside the heart of a database system via a traditional procedural programming language, both of these tasks are naturally specified in a declarative query language. To perform cost estimation, System R requires data statistics like relation cardinalities and index selectivities, which can be packaged into a relational format, and thereby accessible in the Overlog language.

We focus on the basic dynamic programming algorithm for the state-space enumeration at the heart of the System R optimizer. A sketch of the System R dynamic program is given in Figure 5.1, which searches for an optimal plan from a set of query predicates ($PREDS$). We focus here on the search strategy, which enumerates query plans for increasingly-large subgoals of the query. It fills in a dynamic programming table (i.e., *bestplan* array) with the best plans that cover a given number of (relational algebra) predicates. Each entry in this table contains the set of lowest-estimated-cost query plans among all plans producing an *equivalent* output relation (i.e., plans composed of the same predicates), and among the plans that produce an "interesting order." If the plan produces tuples in an order that is relevant to a later join condition, or an "group/order by" clause, then it is considered to be an *interesting order* [82].

The `optimize` procedure in Figure 5.1 takes the set of predicates mentioned in

```
def optimize (PREDS)
 1: Let AM = ∅ be a set of single table access method plans
 2: for all relations r ∈ PREDS do
 3:    AM = AM ⋃ access methods on r
 4: end for
 5:
 6: GRP_AM = GroupBy(f_equivalent, AM)
 7: GRP_AM = GRP_AM − {uninteresting ordered, suboptimal groups ∈ GRP_AM}
 8: bestplan[1] = ArgMin(f_cost, GRP_AM) /* best plans of size 1, from each group */
 9: BP = search (bestplan, PREDS, f_sizeof(PREDS))
10: bp = ArgMin(f_cost, BP) /* best overall plan */
11:
12: if query contains a group by or order by clause then
13:    bop = best ordered plan relative to the clause attributes
14:    return Min(f_sort?(bp), f_sort?(bop)) /* Note: ignores hash grouping plans */
15: else
16:    return bp
17: end if
end

/* Returns a set containing the best size k plans. */
def search (bestplan, PREDS, k)
 1: if bestplan[k] = ∅ then
 2:    /* Get the set of size k − 1 best plans. */
 3:    BP_{k−1} = search (bestplan, PREDS, k − 1)
 4:    Let P_k = ∅ be a set of size k plans
 5:    for all plans bp ∈ BP_{k−1} do
 6:       for all predicates p ∈ PREDS ∉ bp do
 7:          M_k = all methods (e.g., join) that take plan bp (outer) and include p (inner)
 8:          P_k = P_k ⋃ M_k
 9:       end for
10:    end for
11:
12:    /* Group by equivalent plan, and retain optimal and interesting ordered plans. */
13:    GRP_k = GroupBy(f_equivalent, P_k)
14:    GRP_k = GRP_k − {uninteresting ordered, suboptimal groups ∈ GRP_k}
15:
16:    /* The set of size k best plans from each group in GRP_k */
17:    bestplan[k] = ArgMin(f_cost, GRP_k)
18: end if
19: return bestplan[k] /* The set of size k best plans */
end
```

Figure 5.1: Sketch of the System R optimizer algorithm. The `optimize` procedure is called with all predicates mentioned in the query ($PREDS$), while the `search` procedure enumerates the plan space (bottom-up). Each enumeration step generates plans size $k \in [1, \ldots, |PREDS|]$, and stores the set of optimal plans in the *bestplan* array.

the query, and returns an optimal plan to the query. The search begins with plans of size one, which consists of the access methods to any relations mentioned in the query. Note that in P2 the initial plan (of size one) is the event predicate, which is assigned to the rule by the delta rewrite (Chapter 3.2). The event predicate is used to initialize the optimization described in Chapter 5.2.1, instead of the traditional approach; shown here as the optimal table access methods. The `search` procedure captures the essence of generating *plans* of size k, and pruning away those *plans* that are not optimal, nor interesting. The `optimize` procedure makes the "top-level" call to `search`, requesting the best plans that cover all predicates in the query. The `search` returns a reference to this set of "top-level" optimal plans; including those with interesting orders. If the query contains a `group by` or `order by` clause, then we may require a further sorting operation [1] — the cost of which depends on the order of the chosen optimal plan. In the absence of any ordering constraints, we simply return the overall lowest-estimated-cost plan.

In the System R optimizer, the *principle of optimality* is assumed to hold: the lowest-cost solution to some plan is constructed from the optimal solutions to subplans. Thus dynamic programming can proceed in a "bottom-up" fashion. For a given set of predicates ($PREDS$), the optimizer generates plans of size k terms by appending a single (unused) term from $PREDS$ to an optimal plan of size $k-1$ terms, as shown in the loop of `search` procedure of Figure 5.1. There are a few additional details that we have chosen to gloss over in the pseudocode. For instance, avoid combining a k-way plan with a 1-way plan if there is no join condition between them, unless all other predicates with join conditions have been used (i.e., postpone Cartesian products). We handle this case in our Overlog rules by ensuring the cost of a "cross-product" plan is greater than any other plan that contains joining attributes.

We now turn to the description of our Overlog rules for plan generation and conclude with our rules for best plan selection. Our declarative optimizer adds two new tables (`plan` and `bestPlan`) to the Metacompiler Catalog. The `plan` table identifies a join method for evaluating a subgoal as the "inner" relation. Each `plan` tuple contains an identifier, which the `bestPlan` table uses to reference optimal plans. For a given rule body term, the *Planner* stage generates a physical dataflow plan based on the position and join method assigned in the relevant term relation (i.e., `sys::predicate`, `sys::assign` and `sys::select`). Chapter 5.2.1 presents our System R rules for generating plans (`plan` tuples) from the predicates in the rule body. Our rules for selecting a best plan are described in Chapter 5.2.2, which also includes a description of how we estimate selectivities. We then conclude with our termination rules in Chapter 5.2.3.

5.2.1 Plan Generation

Figure 5.1 describes the System R algorithm in two phases; access method plan generation and plan enumeration for increasingly large subgoals. Recall from Chapter 3.2 that P2 converts a rule into an event-condition-action (ECA) form, where the event

[1] This pseudocode ignores hashing plans for `group by`.

```
sr1 plan(@A, Pid, Rid, PlanID, Group, Sort, Schema, Card, Cost) :-
    systemr::programEvent(@A, Pid, ...),
    sys::rule(@A, Pid, Rid, ...),
    sys::predicate(@A, Pid, Rid, PredID, ..., Schema, Pos, ...),
    Pos == 1,
    PlanID := f_cons(``delta'', PredID),
    Group := f_cons(PredID, null),
    Sort := null,
    Card := 1, Cost := 1.
```

Figure 5.2: Plan seed rule.

predicate represents a stream of tuples representing side-affect actions (i.e., insert and delete) to the reference table. As a consequence of this dataflow design, our first phase simply generates a plan that listens for such event tuples. The reader can assume the delta rewrite stage executes before the System R optimizer stage, and that the delta predicate is in the first rule position.

Figure 5.2 contains the single rule that creates an initial plan, from each rule in the program, using the delta predicate. A `plan` tuple represents a query plan for a given rule, and the plan's `size` reflects the number of term identifiers covered in the *Group* variable (i.e., the number of leaves in the plan tree). The optimizer listens on the `systemr::programEvent` event stream in rule `sr1`, which initiates the optimization process. The `systemr::programEvent` tuple is joined with the `sys::rule` table along the *Pid* (program identifier) variable to obtain the set of rules defined in the input program. This result set of rule tuples is joined with the `sys::predicate` table along the *Rid* (rule identifier) variable; producing a tuple for each predicate term defined by a given rule. The predicate term assigned to position 1 ($Pos == 1$) is by convention the event predicate term. The result of this rule creates a `plan` of "size one" for each rule in the input program. The *Group* variable is initialized to a list containing the *PredID* of the event predicate and the *PlanID* is used to hold the actual plan definition. As plan enumeration proceeds, we append new subgoal term identifiers to the *Group* variable and physical operator descriptions (e.g., sort-merge join) to the *PlanID* variable.

The Overlog optimizer defines a set of plan generation rules that together perform the induction step of the dynamic program. These rules extend a best plan of k terms with a $(k+1)^{st}$, thus far unused term from the rule body. If the new term considered is a table predicate, then the new plan (*PlanID*) is annotated with an appropriate join method, which takes the optimal subplan and "joins it" with the predicate table. The join methods supported by P2 include scanned and index-nested-loop join, as well as sort-merge join. A plan tuple also carries with it an associated cost, which only considers CPU costs since all P2 relations reside in memory. [2] We now turn

[2]Including other cost metrics (e.g., I/O) would entail modifying the cost estimations defined in rules `sr2`, `sr3`, and `sr4`.

```
sr2 plan(@A, Pid, Rid, PlanID, Group, Sort, Schema, Card, Cost) :-
    bestPlan(@A, Pid, Rid, OPlanID),
    plan(@A, Pid, Rid, OPlanId, OGroup, OSort, OSchema, OCard, OCost),
    sys::predicate(@A, Pid, Rid, PredID, ..., Tid, PSchema, Pos, ...),
    Pos > 1,
    sys::table(@A, Tid, ..., TCard, Sort),
    f_contains(PredID, OGroup) == false,
    PlanID := f_cons(''nested-loop'', OPlanId, PredID),
    Group  := f_cons(PredID, OGroup),
    Schema := f_joinSchema(OSchema, PSchema),
    Sort   := OSort,
    Card   := f_nlj_card(OCard, OSchema, TCard, PSchema),
    Cost   := f_nlj_cost(OCost, OSchema, TCard, PSchema).
```

Figure 5.3: nested-loop join method.

to the description of the rules that generate plans for nested-loop-join, index nested-loop-join, and sort-merge join methods.

All materialized table predicates appearing in the rule body are considered when creating a nested-loop join plan, which is derived by rule `sr2` in Figure 5.3. Rule `sr2` is evaluated on an update to the `bestPlan` relation (described in Chapter 5.7), which contains the plan identifier ($OPlanID$) used to select the reference (optimal) subplan in the `plan` relation. The result of joining `bestPlan` with the `plan` table gives us the "outer" plan of the nested-loop join method.

We extend the "outer" `plan` with an "inner" table predicate by joining with the `sys::predicate` relation along the same rule identifier (Rid). The selection predicate $Pos > 1$ ensures that we do not consider the rule head predicate (the zeroth term by convention) or the delta predicate (the first term position). The outer plan tuple contains a list ($OGroup$) of the term identifiers that already appear in it. This list is used to prune results that reference inner table predicates already appearing in the outer plan. This test happens in the $f\_contains$ function, which checks for inner table predicate membership in the outer plan term list.

The next step is to assign a cost to our nest-loop join plan. This cost depends on cardinality estimates for the outer plan — already defined in the `plan` tuple — and the inner relation. Cardinality estimates for the inner relation are given by the `sys::table` predicate, which is joined with the `sys::predicate` in rule `sr2` along the Tid (table identifier) variable. The functions f_nlj_cost and f_nlj_card consider existing costs and cardinality estimates, as well as the (join) input schemas. If the input schemas force a cross-product plan, then f_nlj_cost assigns an infinite cost, which postpones this plan relative to other plans that contain joining attributes. We also note that the result order that this plan produces is identical to the order of the outer plan, which is referenced by the $OSort$ variable.

An index-nested-loop join plan is generated by rule `sr3` in Figure 5.4. Like rule `sr2`, it joins the `bestPlan`, `plan`, `sys::predicate`, and `sys::table` predicates to get all

```
sr3 plan(@A, Pid, Rid, PlanID, Group, Sort, Schema, Card, Cost) :-
    bestPlan(@A, Pid, Rid, OPlanID),
    plan(@A, Pid, Rid, OPlanId, OGroup, OSort, OSchema, OCard, OCost),
    sys::predicate(@A, Pid, Rid, PredID, ..., Tid, PSchema, Pos, ...),
    Pos > 1,
    sys::table(@A, Tid, ..., TCard, Sort),
    sys::index(@A, Iid, Tid, Key, Type, Selectivity),
    f_contains(PredID, OGroup) == false,
    f_indexMatch(OSchema, PSchema, Key),
    PlanID := f_cons(''index-loop'', OPlanID, PredID, Iid),
    Group  := f_cons(PredID, OGroup),
    Sort   := OSort,
    Card   := OCard * (Selectivity * TCard),
    Cost   := OCost + Card.
```

Figure 5.4: index-nested-loop join method.

```
sr4 plan(@A, Pid, Rid, PlanID, Group, Sort, Schema, Card, Cost) :-
    bestPlan(@A, Pid, Rid, OPlanID),
    plan(@A, Pid, Rid, OPlanId, OGroup, OSort, OSchema, OCard, OCost),
    sys::predicate(@A, Pid, Rid, PredID, ..., Tid, PSchema, Pos, ...),
    Pos > 1,
    sys::table(@A, Tid, ..., TCard, TSort),
    f_contains(PredID, OGroup) == false,
    JM     := f_sortPlan(OSort, OSchema, PSchema, TSort),
    PlanID := f_cons(''sort-merge'', OPlanID, PredID, JM),
    Group  := f_cons(PredID, OGroup),
    Sort   := f_sortJoinAttributes(OSort, OSchema,
                                   PSchema, TSort),
    Schema := f_sortMerge(Sort, OSchema, PSchema),
    Card   := OCard * (TCard / 10),
    Cost   := f_sortCost(JM, OCard, TCard).
```

Figure 5.5: sort-merge join method.

table predicates and cardinality estimates for predicates that do not appear in the *OGroup* term list. That result is subsequently joined with the (additional) `sys::index` predicate, which adds index information to the this result. The function *f_indexMatch* tests if the index can be used to perform the join using attributes from the outer plan schema (*OSchema*) and attributes from the inner predicate table (*PSchema*). Any resulting tuples are assigned (example) cardinality and cost estimates, which now use the additional *index* selectivity information given by the *Selectivity* variable defined by the `sys::index` predicate. We also support range predicates in our index-nested-loop join plans but do not discuss them in detail.

Figure 5.5 shows the rule for generating a sort-merge join plan, which considers a best plan and a new table predicate joined along some ordered attributes. The tuples from the outer plan and the inner table predicate can be ordered by some attributes,

```
sr5 plan(@A, Pid, Rid, PlanID, Group, Sort, Schema, Card, Cost) :-
    bestPlan(@A, Pid, Rid, OPlanID),
    plan(@A, Pid, Rid, OPlanId, OGroup, OSort, OSchema, OCard, OCost),
    sys::select(@A, Sid, Rid, BoolExpr, ...),
    f_contains(Sid, OGroup) == false,
    f_filter(OSchema, BoolExpr) == true,
    PlanID := f_cons(``filter'', OPlanID, Sid),
    Group  := f_cons(Sid, OGroup),
    Sort   := OSort,
    Schema := OSchema,
    Cost   := OCost,
    Card   := OCard / 3.
```

Figure 5.6: selection predicate filter plan.

or not. We note that the *TSort* variable in the `sys::table` table identifies the ordered attributes of the inner relation, while *OSort* refers to the order of the outer tuples.

The join method variable *JM* is given a value that indicates the need to presort the inner relation, or not. In our implementation of the sort-merge join operator, we decided not to sort the outer relation by first draining all of its tuples, sorting them, and then merging with the sorted inner relation. [3] Instead, each outer tuple is used to perform a binary search on the sorted inner relation, which returns any tuples that join along the relevant attributes. If we know that the tuples from the outer result will be given in order, then we can optimize this binary search to be like a merge-join. [4] These costs are considered by the *f_sortCost* function, which takes the assigned join method and the input cardinalities and returns a plan cost. The output of a sort-merge join plan includes the join attribute in the *Sort* variable.

Figure 5.6 contains a rule that creates a plan out of any selection predicates in the rule body. A selection predicate plan is created when all variables mentioned in its boolean expression (*BoolExpr*) are bound by the current outer plan schema (*OSchema*). Applying a selection filter does not change the sorting attribute of the outer plan, nor does it effect its schema. We assume the cost of a "filter" plan is negligible, but could add a function that considers certain operational costs. Furthermore, we use a generic cardinality estimation here but could associate meta-data (e.g., attribute distributions and min/max values) with the `plan` relation that would tune this estimator.

5.2.2 Best plan selection

Figure 5.7 shows the rules that select the best plan from a set of equivalent plans, in terms of the output they produce and the order in which it comes. The `bestGroupCost`

[3] This would have added significant complexity to the P2 dataflow architecture, which is optimized for tuple at a time processing.

[4] We maintain a cursor state on the inner relation that tells us where the last join match occurred.

```
sr6  bestGroupCost(@A, Pid, Rid, Group, a_min<Cost>) :-
     plan(@A, Pid, Rid, PlanID, Group, ..., Cost).

sr7  bestOrderCost(@A, Pid, Rid, Group, Sort, a_min<Cost>) :-
     interestingOrder(@A, Pid, Rid, PlanID),
     plan(@A, Pid, Rid, PlanID, Group, Sort, ..., Cost).

sr8  interestingOrder(@A, Pid, Rid, PlanID) :-
     plan(@A, Pid, Rid, PlanID, ..., PlanSchema, ..., Cost),
     sys::rule(@A, Pid, Rid, HeadPredID, ...),

     /* The head predicate */
     sys::predicate(@A, Pid, Rid, HeadPredID, ..., HeadPredSchema, ...),

     /* A rule body predicate */
     sys::predicate(@A, Pid, Rid, BodyPredID, ..., BodyPredSchema, ...),
     HeadPredID != BodyPredID,

     /* participates in a later join OR
        is a prefix of a grouping attribute */
     (f_contains(BodyPredID, PlanID) == false &&
      f_contains(f_joincond(PlanSchema, BodyPredSchema), Sort)) ||
     f_isGroupByPrefix(Sort, HeadPredSchema) == true.

sr9  bestPlan(@A, Pid, Rid, PlanID) :-
     bestGroupCost(@A, Pid, Rid, Group, Cost),
     plan(@A, Pid, Rid, PlanID, Group, Sort, ..., Cost),

sr10 bestPlan(@A, Pid, Rid, PlanID) :-
      bestOrderCost(@A, Pid, Rid, Group, Sort, Cost),
      plan(@A, Pid, Rid, PlanID, Group, Sort, ..., Cost),
```

Figure 5.7: Best plan selection.

predicate of rule `sr6` identifies the plan with the minimum cost from the set of equivalent plans, regardless of order. This is followed by rule `sr7`, which queries the `plan` and `interestingOrder` relations for the minimum cost plans for each equivalent interesting order. Recall that the *Group* variable references all the predicate identifiers that participate in this plan. We use a *set-based* container object to hold these identifiers so that when a comparison is made between two such objects, it is based on equivalent plans. Therefore, the purpose of the *Group* variable is to ensure that we select the minimum cost plan among the *set of* equivalent plans. The purpose of rule `sr6` is to ensure we consider the costs associated with interesting ordered plans.

Rule `sr8` determines if a plan, ordered by some given attributes, is interesting. This occurs in P2 when the plan is sorted along attributes that are relevant to a later join or are a prefix of grouping attributes. The body of this rule joins a `plan` tuple with the `predicate` table, twice, to get the head predicate and a body predicate that does not already exist in the plan. The final selection predicate in this rule checks the necessary conditions, and if met, the rule will generate an `interestingOrder` tuple referencing the given $PlanID$. The remaining two rules (`sr9` and `sr10`) populate the `bestPlan` table with the actual optimal plan information.

Improving Selectivity Estimation

For equality selection predications, our System R rules above support selectivity estimates using a uniform distribution estimator given by the index. For more precise estimates and to handle range predicates, we have defined declarative rules that produce equiwidth histograms (*ew-histograms*); additional histogramming rules could be added analogously. The creation of an ew-histogram is triggered by the installation of a fact in a metadata table of the ew-histograms defined in the system. The metadata table contains the parameters of the histogram (i.e., the table name, the attribute position, and the number of buckets). For example, the fact

$$\texttt{sys::ewhistogram::metadata}(@LOCALHOST, \text{"}pred\text{"}, 3, 10).$$

creates a ten bucket equi-width historgram on table `pred` for the attribute in the third position.

Each fact in the ew-histogram table triggers Evita Raced rules that themselves generate new rules to create ew-histograms (determining bucket boundaries based on the bucket count and the min and max values of the attribute), and to maintain bucket counts (performing a count aggregation over the table attributes, grouped by the bucket boundaries). The compiler stage that generates ew-histograms in this fashion consists of 23 rules (92 lines). The histogram data is stored in relational format with each row corresponding to a single bucket. Exploiting these histograms required an aggregation query to sum up appropriate bucket boundaries based on selection predicates in the user query. The cost and selectivity estimators, in the `plan` generation rules, were then modified to incorporate the result of these bucket aggregates, and used to obtain density estimations for a given selection predicate

```
sr11 rules(@A, Pid, a_count<Rid>) :-
     systemr::programEvent(@A, Pid, ...),
     sys::rule(@A, Pid, Rid, ...).

sr12 completeRule(@A, Pid, Rid) :-
     bestPlan(@A, Pid, Rid, PlanID),
     sys::rule(@A, Pid, Rid, ..., Goals),
     f_sizeof(PlanID) == Goals - 1.

sr13 completeRuleCount(@A, Pid, a_count<Rid>) :-
     completeRule(@A, Pid, Rid).

sr14 sys::program(@A, Pid, ..., ''systemr'', ...) :-
     completeRuleCount(@A, Pid, Count),
     rules(@A, Pid, Count),
     sys::program(@A, Pid, ..., Stage, ...).
```

Figure 5.8: System R termination rules.

5.2.3 Termination

Figure 5.8 presents our rules for terminating the System R optimizer stage. Rule `sr11` counts the number of rules in the target program. This count will be used to check for our end condition, which occurs when all rules have been given a `bestPlan` tuple with a plan size equal to the number of subgoals. Rule `sr12` identifies the completion of a rule based on this end condition, while rule `sr13` counts the number of completed rules for a given program. Finally, when the counts in `completeRuleCount` and `rules` are equal (a familiar pattern), rule `sr14` generates the termination signal for a given program by inserting a new tuple into the `program` program with the "systemr" stage name.

5.3 Cascades Optimization

The bottom-up, dynamic programming search strategy described in Chapter 5.2 is a natural fit to a Datalog-based rule language. One might think a top-down Cascades-style optimization strategy [36] would be difficult to implement since Overlog, like Datalog, is evaluated in a bottom-up fashion. This is partially true but still relatively straightforward. Since the System R search strategy conforms to the Overlog evaluation strategy, we did not need to write explicit rules for traversing through the plan space. That is, the System R search strategy was implicitly implemented by the Overlog bottom-up evaluation. A top-down search strategy, on the other hand, requires extra logic to guide the search through the plan space in a top-down order. The logic of a top-down search strategy follows a dynamic programming technique called memoization, which turns out to be just as natural and intuitive in Overlog, and therefore can be implemented in Evita Raced.

The remainder of this chapter presents our implementation of the Cascades branch-and-bound optimization in Overlog. Chapter 5.3.1 provides a short description of the Cascades algorithm, before describing our declarative rules that implement the algorithm. Our rules are divided into three logical modules — search strategy (Chapter 5.3.2), plan generation (Chapter 5.3.3) and winner selection (Chapter 5.3.4) — that model a paper description [86]. Our rules for plan generation and winner selection may remind the reader of the plan generation and best plan rules in the previous System R discussion. However, the search strategy rules are unique to this optimization stage, and will therefore be the focus our attention.

5.3.1 Overview

Our description of the Cascades optimizer follows the notation of Shapiro, et al. [86]. Cascades' plans are classified into *groups*, which is an equivalence class of expressions (i.e., predicates) that produce the same result. During the optimization, each group (e.g., [ABC] consisting of table predicates A, B, and C) represents a container to physical plans (e.g., {[AB] `sort-merge-join` [C]}, {[B] `nested-loop-join` [AC]}, ...) over subexpressions in that group. In order to keep the search space small, a group only references top-level physical plans through *multiexpressions*, which are plan expressions that restrict the input of operators to subgroups. For example, group [ABC] references the multiexpression {[AB] `sort-merge-join` [C]}, whose `sort-merge-join` operator takes groups [AB] and [C] as input, instead of the (possibly many) individual plans within these subgroups. Associated with each group is a *winner's circle*, which identifies the optimal plan within a given group, and will be the plan chosen to represent the group, referenced by top-level multiexpressions.

At a high-level, the branch-and-bound algorithm that drives the Cascades optimizer performs the following actions. The search strategy generates groups in a top-down order, and within each group it performs a bottom-up search for the cheapest multiexpression, which is called the *winner*. The top-down order follows a depth-first search over the space of multiexpressions, where a particular *branch* (multiexpression) is fully explored before considering another. An upper bound, initialized to ∞, is assigned to each group. The upper bound is updated as new (cheaper) multiexpressions for the given group are discovered. The group bound is carried down each branch of the depth-first search. A multiexpression is pruned if its cost exceeds the group bound. The optimization terminates when the root group (containing all expressions in the query) has been fully explored, and a winner chosen. In the discussion that follows, when we indicate a *plan* we mean a multiexpression within a group.

5.3.2 Search strategy

The optimization begins when the root group (e.g., [ABC]) is inserted into the `group` table, and a `branch` tuple is created to initiate a depth-first traversal over the plan space. This is initiated by rules `bb1` and `bb2` in Figure 5.9 after the

```
/* Initialize the top-level group */
bb1 groupSeed(@A, Rid, a_list<PredID>, a_list<Schema>) :-
    cascades::programEvent(@A, Pid, ...),
    sys::predicate(@A, Pid, Rid, PredID, ..., Schema, Pos, ...),
    Pos > 0. // Exclude the head predicate

bb2 group(@A, Rid, GroupID, PredList, SchemaList) :-
    groupSeed(@A, Rid, PredList, SchemaList),
    GroupID := f_mkGroupID(PredList).

/* Initialize a new branch and bound on the given group. */
bb3 branch(@A, Rid, GroupID, Pos, Bound) :-
    group(@A, Rid, GroupID, PredList, SchemaList),
    Pos   := 0,
    Bound := infinity.

/* Subgroup with all predicates except position Pos */
bb4 group(@A, Rid, SubGroupID, SubPredList, SubSchemaList) :-
    branch(@A, Rid, GroupID, Pos, Bound),
    group(@A, Rid, GroupID, PredList, SchemaList),
    Pos < f_sizeof(PredList),
    SubPredList   := f_remainder(PredList, Pos),
    SubSchemaList := f_remainder(SchemaList, Pos),
    SubGroupID    := f_mkGroupID(SubPredList).

/* Subgroup with only the predicate at position Pos */
bb5 group(@A, Rid, SubGroupID, SubPredList, SubSchemaList) :-
    branch(@A, Rid, GroupID, Pos, Bound),
    group(@A, Rid, GroupID, PredList, SchemaList),
    Pos < f_sizeof(PredList),
    SubPredList   := f_get(PredList, Pos),
    SubSchemaList := f_get(SchemaList, Pos),
    SubGroupID    := f_mkGroupID(SubPredList).

/* Move the branch position forward when the branch group is complete. */
bb6 branch(@A, Rid, GroupID, Pos+1, Bound) :-
    branchComplete(@A, Rid, GroupID, Pos, Bound).
```

Figure 5.9: Cascades top-down search strategy rules.

`cascades::programEvent` tuple is received. Rule `bb1` aggregates *lists* (not sets) of predicate identifiers and schemas, for each rule in the program. Rule `bb2` converts `groupSeed` tuples to `group` tuples by including a $GroupID$ variable, which is initialized to a set-based object containing the identifiers in the $PredList$ variable.

Rule `bb3` triggers on an update to the `group` relation, creating a `branch` tuple with the given group identifier, an initial branch position, and an initial group bound (∞). Rules `bb4` and `bb5` create new subgroups; first (`bb4`) by excluding the predicate at the given branch position Pos, and second (`bb5`) by including just that branch position's predicate. As an aside, these two rules would need to be modified in order to consider "bushy" plans. As we will see in Chapter 5.3.3, `branch` tuples are used for generating `plan` tuples. Here, we must ensure that the plan enumeration does not update the branch position until all plans relevant to that position have been discovered. We detect this condition in rule `bb6` with the `branchComplete` predicate: described in Chapter 5.3.4.

5.3.3 Plan Generation

Figure 5.10 presents two rules for generating `plan` tuples relevant to a particular branch position. Rule `bp7` handles the case when a single predicate identifier is referenced in the $GroupID$ value. The plan in this case is a streaming delta predicate, which is placed in the first position of the rule body.

Rule `bp8` generates a nested-loop join `plan` using a "winner" plan as the outer and a single table predicate as the inner. The `winner` relation (described in Chapter 5.3.4) identifies the best plans — including interesting orders — for a given group. The rule joins the `winner` predicate with the `plan` predicate to obtain an actual (best) plan. For the inner predicate, we look for a `branch` containing a single predicate then, using the `sys::predicate` and `sys::table` predicates, we obtain the desired inner information (i.e., $PredID$, $ISchema$, and $TCard$ variables). The parent `branch` is identified by equating the $GroupID$ to the combined child identifiers: given by $OGroupID$ and $IGroupID$. The parent `branch` provides the $Bound$ variable, which is used here to prune expensive plans. The final step in this rule creates the remaining variables needed to project onto the `plan` predicate. Like our System R rules, we use the plan identifier ($PlanID$) to hold the actual plan definition.

The rules that cover index-loop and sort-merge join methods trivially follow from rule `bb8` and the respective System R rules `sr3` and `sr4`, so we elide their details. Like the System R rules, we need to consider various properties of these join methods. First, when considering an index-loop join, we include the index definition relevant to the joining attributes. Second, for a sort-merge join, we ensure the join attributes define the order of the inner relation — sorting if necessary — so that for each tuple in the outer plan, we can perform a (possibly optimized) binary search on inner relation. The plan cost depends on the ordering properties of the inputs and the plan output is ordered by the join attributes. We also omit the rule that handles selection predicates, which resembles rule `sr5` in Figure 5.6.

```
bp7 plan(@A, Rid, GroupID, PlanID, Schema, Sort, Card, Cost) :-
    branch(@A, Rid, GroupID, Pos, Bound),
    group(@A, Rid, GroupID, _, SchemaList),
    f_sizeof(PredList) == 1,
    PlanID := f_cons(``delta'', f_get(GroupID)),
    Schema := f_mkSchema(SchemaList),
    Sort   := null,
    Card   := 1,
    Cost   := 1.

bp8 plan(@A, Rid, GroupID, PlanID, Schema, OSort, Card, Cost) :-
    /* Information associated with some winner subplan. */
    winner(@A, Rid, OGroupID, OPlanID),
    plan(@A, Rid, OGroupID, OPlanID, OSchema, OSort, OCard, OCost),

    /* Evaluate predicates that belong to a branch of size one */
    branch(@A, Rid, IGroupID, _, _),
    f_sizeof(IGroupID) == 1, // contains a single predicate
    sys::predicate(@A, Pid, Rid, PredID, ..., Tid, ISchema, Pos, ...),
    f_get(IGroupID) == PredID,
    f_exists(PredID, OGroupID) == false, // not part of outer plan
    sys::table(@A, Tid, ..., TCard, TSort),

    /* Find the parent branch */
    branch(@A, Rid, GroupID, Pos, Bound),
    f_combine(OGroupID, IGroupID) == GroupID,

    PlanId := f_cons(``nested-loop'', OPlanID, PredID),
    Schema := f_mkSchema(OSchema, ISchema),
    Card   := f_card(OSchema, ISchema, OCard, TCard),
    Cost   := f_cost(OSchema, ISchema, OCost, OCard, TCard),
    Cost <= Bound.
```

Figure 5.10: Cascades plan generation rules for event predicates and nested-loop join method.

5.3.4 Winner Selection

The rules in Figure 5.11 select *winner* plans from the plans generated for a given group. We begin with rule `bb9`, which determines the cost of an optimal plan, for each group, regardless of its order. Rule `bb10` does the same but also considers interesting orders. Rule `bb11` is nearly identical to rule `sr8`; both determine the orders that are interesting based on later grouping and joining attributes. Finally, rules `bb12` and `bb13` select winners based on the costs referenced by the `bestGroupCost` and `bestOrderCost` predicates. Note that we only generate a winner plan after we have fully explored a branch; the `Pos == f_sizeof(GroupID)` predicate ensures this constraint. The need for this explicit constraint is due to the lack of stratification support in P2.

Rule `bb14` serves as a feedback loop to the search strategy rule `bb6`, which moves the `branch` position forward by one after fully exploring the current branch. A branch is fully explored when winners have been derived along both child branches. Rule `bb14` detects this case with the `f_isChildBranch(...)` function, which uses the *PredList* and (branch) *Pos* variables associated with the parent group to evaluate the group identifiers belonging to the two child winners. The rule also updates the branch bound with the cost given in the `bestGroupCost` predicate, relative to the parent branch identifier. We note that the rule `bb9` is not predicated on the `winner` predicate relative to the parent branch so it can be used to provide the latest best cost value. Furthermore, a rule, similar to `bb14`, considers cost bounds from `bestOrderCost` by ensuring the lowest overall cost is used to bound subsequent branches.

5.3.5 Termination

Figure 5.12 contains the four rules used to detect the termination condition of this optimization stage. These rules resemble the System R termination rules in Figure 5.8. The first rule (`bb15`) counts the total number of rules in the target program. Rules `bb16` and `bb17` count how many rules have completed, which occurs when the `branch` cursor has moved beyond the last predicate. Some number of fixpoints later, when the `completeRuleCount` reaches the total number of rules in the program, rule `bb18` terminates the optimization stage, and projects a new `sys::program` tuple with the stage attribute set to "cascades."

```
/* Determine the best overall cost for a given plan. */
bb9 bestGroupCost(@A, Rid, GroupID, a_min<Cost>) :-
      plan(@A, Rid, GroupID, _, _, _, _, Cost).

/* Determine the best cost plan for each ordered result. */
bb10 bestOrderCost(@A, Rid, GroupID, Sort, a_min<Cost>) :-
      plan(@A, Rid, GroupID, _, _, Sort, _, _),
      interestingOrder(@A, Pid, Rid, PlanID).

/* Identify interesting ordered plans. */
bb11 interestingOrder(@A, Pid, Rid, PlanID) :-
      plan(@A, Rid, GroupID, PlanID, Schema, Sort, _, _),
      sys::rule(@A, Pid, Rid, HeadPredID, ...),
      sys::predicate(@A, Pid, Rid, HeadPredID, ..., HeadPredSchema, ...),
      sys::predicate(@A, Pid, Rid, BodyPredID, ..., BodyPredSchema, ...),
      HeadPredID != BodyPredID,
      (f_contains(BodyPredID, GroupID) == false &&
       f_contains(f_joincond(Schema, BodyPredSchema), Sort)) ||
      f_isGroupByPrefix(Sort, HeadPredSchema) == true.

/* Choose a winner based on the best overall cost. */
bb12 winner(@A, Rid, GroupID, PlanId) :-
      bestGroupCost(@A, Rid, GroupID, Cost),
      branch(@A, Rid, GroupID, Pos, _),
      Pos == f_sizeof(GroupID), // Ensures a fully explored branch
      plan(@A, Rid, GroupID, PlanID, ..., Cost).

/* Choose a winner from each interesting ordered plans. */
bb13 winner(@A, Rid, GroupID, PlanId) :-
      bestOrderCost(@A, Rid, GroupID, Sort, Cost),
      branch(@A, Rid, GroupID, Pos, _),
      Pos == f_sizeof(GroupID), // Ensures a fully explored branch
      plan(@A, Rid, GroupID, PlanID, _, Sort, _, Cost).

/* branchComplete: when both child branches have winners */
bb14 branchComplete(@A, Rid, ParentGroupID, Pos, Bound) :-
     winner(@A, Rid, ChildGroupID1),
     winner(@A, Rid, ChildGroupID2),
     branch(@A, Rid, ParentGroupID, Pos, Bound),
     group(@A, Rid, ParentGroupID, PredList, _),
     f_isChildBranch(PredList, Pos, ChildGroupID1, ChildGroupID2),
     bestGroupCost(@A, Rid, ParentGroupID, Cost),
     Bound := Cost < OldBound ? Cost : OldBound.
```

Figure 5.11: Cascades winner selection rules.

```
bb15 rules(@A, Pid, a_count<Rid>) :-
     cascades::programEvent(@A, Pid, ...),
     sys::rule(@A, Pid, Rid, ...).

bb16 completeRule(@A, Pid, Rid) :-
     branch(@A, Rid, GroupID, Pos, _),
     f_sizeof(GroupID) == Pos.

bb17 completeRuleCount(@A, Pid, a_count<Rid>) :-
     completeRule(@A, Pid, Rid).

bb18 sys::program(@A, Pid, ..., ``cascades'', ...) :-
     completeRuleCount(@A, Pid, Count),
     rules(@A, Pid, Count),
     sys::program(@A, Pid, ..., Stage, ...).
```

Figure 5.12: Cascades termination rules.

Chapter 6

Evita Raced: Declarative?

When we started this work, the vision of declaratively specified query optimization was appealing thanks to its elegance and its promise of usability and maintainability. Although we remain convinced on this front, our optimism was tempered by the pragmatics of developing software within a continuously changing system prototype. Here we reflect on some of the (hard) lessons we learned while conducting this research.

6.1 A Candid Reflection

P2's notion of consecutive Datalog-style fixpoints, especially in networked environments, still had many rough edges, both on the design and on the engineering front. Because deep down P2's runtime is an event-driven execution engine, its basic unit of atomicity was akin to a single iteration through a recursive query evaluation strategy like seminaïve evaluation, generating a set of derived actions (tuples to be inserted, deleted, transmitted remotely, or evaluated locally for further deduction) from a single incoming event, and committing changes to the database atomically upon completion of such a step [64]. P2's Datalog-style fixpoints were implemented as sequences of such single-event iterations. As a result, the system's design shares both event-driven and logic-style flavors, with some unresolved conflicts (e.g., stratified Datalog).

Second, as in most prototypes, the programmer interface was not polished. Debugging was difficult, especially since the logic language made it tough to understand which value corresponded to which formal attribute in a long tuple of a dozen or more attributes. Though concise, declaratively specified optimizations pack a punch in terms of density of concepts, which only becomes deadlier due to the (otherwise desirable) arbitrary order of rule execution. Certainly a better thought-out system to debug declarative programs – optimizations, no less – would have made the job easier. To be fair, however, our experience with building monolithic optimizers in production database management systems in the past was not a great deal rosier. It is hard to debug code when the output's correctness (e.g., minimality of cost) is too expensive to verify.

Third, the evolution of the Overlog language had a long way to go. The P2 version of the language offered no modularity, making it tough to isolate and reuse logically distinct components. It did have a rudimentary concrete type system, but had poor support for structured types like matrices and lists. Overlog in P2 "cut corners" on the proper set-orientation of Datalog; since program stratification was not present in the system, dealing with streaming aggregates required us to resort to imperative tricks like matching "counts", computed in separate "dataflow fixpoints", to determine that state was ready to be finalized.

Beyond particular characteristics of P2, one hard lesson we learned was that extensibility and ease of use at the top often comes at the expense of complexity below the extensibility layer. The tabularization of compiler state to enable declarative optimizations also meant that even imperative compiler stages such as our bootstrap stages implemented in C++ had to use tables, foregoing their familiar interaction with C++ data structures. Building glue libraries to ease this interaction might have relieved this pain.

Nevertheless, despite these complaints, we were able to get all of our desired optimizations expressed in Overlog in a highly compact way, as promised by the various earlier papers on P2. By contrast, the initial version of P2 had no query optimizations of interest beyond localization, which was really a requirement imposed by the P2 dataflow architecture on rules containing distributed predicates.

Finally, the cyclic dataflow used for stage scheduling in Evita Raced (Section 3.1.2) resembles the continuous query engine of TelegraphCQ, with our StageScheduler and Demux elements working together to behave somewhat like the TelegraphCQ *eddy* operator [22]. This connection occurred to us long after we developed our design, but in retrospect the analogy is quite natural: Evita Raced stages are akin to TelegraphCQ's "installed" continuous queries, and P2's Overlog queries are akin to data streaming into TelegraphCQ.

6.2 Conclusion

The Evita Raced metacompilation framework allows Overlog compilation tasks to be written in Overlog and executed in the P2 runtime engine. It provides significant extensibility via a relatively clean declarative language. Many of the tasks of query optimization – dynamic programming, dependency-graph construction and analysis, statistics gathering – appear to be well served by a recursive query language. The notion of metacompilation also leads to a very tight implementation with significant reuse of code needed for runtime processing.

Even with the caveats expressed in Chapter 6.1, we are convinced that a declarative metacompiler is much easier to program and extend than the monolithic query optimizers we have worked on previously. We achieved a point where we could add significant features (e.g., histograms, broadcast rewrites, stratification tests) in an hour or two, where they would otherwise have taken days or weeks of work in a tra-

ditional implementation. One surprising lesson of our work was the breadth of utility afforded by the metacompilation framework. Although motivated by performance optimizations, we have used Evita Raced for a number of unforeseen tasks. These include: automatically expanding user programs with instrumentation and monitoring logic; generating pretty-printers for intermediate program forms; language wrappers for secure networking functionality in the manner of SecLog [5]; stratification detectors and other static code analysis. None of these are performance optimizations per se, but all fit well within an extensible, declarative program manipulation framework. More generally, we believe that metacompilation is a good design philosophy not only for our work, but for the upcoming generation of declarative engines being proposed in many fields.

Chapter 7

BOOM: A Cloudy Beginning

The term "cloud computing" made its mainstream debut in 2007 when companies like Amazon, Google, IBM and Yahoo!, as well as a number of universities, embarked on a large scale *cloud computing* research project [60]. Conceptually, cloud computing is similar to grid computing in terms of multiplexing massive computing resources among a diverse set of applications. The primary difference with cloud computing, over the grid computing model of the 1990s, is its accessibility to the outside world. Today, companies like Amazon, Google and Yahoo! expose parts of their internal computing resources (data centers) to outside developers, using a cost model that is reminiscent of a traditional public utility (i.e., a pay-per-use model). The most prominent example of cloud computing today is the Amazon Elastic Compute Cloud (EC2), which allows users to rent virtual computers to run their applications (e.g., web-server, database).

A challenge moving forward is identifying the right developer API to expose for these large distributed computing platforms. Although today's cloud interfaces are convenient for launching multiple independent instances of traditional single-node services, writing truly distributed software remains a significant challenge. Distributed applications still require a developer to orchestrate concurrent computation and communication across many machines, in a manner that is robust to delays and failures. Writing and debugging distributed system code is extremely difficult even for experienced system architects, and drives away many creative software designers who might otherwise have innovative uses for these massive computing platforms.

Although distributed programming remains hard today, one important subclass is relatively well-understood by programmers: data-parallel computations expressed using interfaces like MapReduce [28], Dryad [48], and SQL. These programming models substantially raise the level of abstraction for programmers: they mask the coordination of threads and events, and instead ask programmers to focus on applying functional or logical expressions to collections of data. These expressions are then auto-parallelized via a dataflow runtime that partitions and shuffles the data across machines in the network. Although easy to learn, these programming models have

traditionally been restricted to batch-oriented computations and data analysis tasks — a rather specialized subset of distributed and parallel computing.

The majority of computations that run in the cloud today are derived from MapReduce workloads. High-level languages like Pig [70], Hive [91], Scope [20] and Jaql [18], all compile down to map and reduce operations. In many regards, MapReduce is considered the programming interface for data-parallel workloads in the "cloud" [10]. The importance of this new computing model led us to look at its most popular open source implementation – Hadoop [71]. We identified parts of the Hadoop system that we thought would benefit from a declarative perspective. We focused on the Hadoop Distributed File System (HDFS), and the Hadoop MapReduce scheduler, which are large system components that support the distributed computation of MapReduce.

The one thing that was rather surprising to us was the code complexity of these system components. The Hadoop MapReduce component (under the `org.apache.hadoop.mapred` package) as of version 18.2 was around 61,183 lines of Java code. The sheer amount of code alone made it difficult to add new features; delaying many requests for new scheduling policies i.e., LATE [104], fair share [43], and capacity scheduler [42].

We explored the cause of such development complexities in the BOOM project; by first developing a declarative implementation of Hadoop and then extending it with new features i.e., alternative scheduling policies. The initial project members included Peter Alvaro, Tyson Condie, Neil Conway, Joseph M. Hellerstein, William Marczak, and Russell Sears. BOOM stands for the *Berkeley Orders Of Magnitude*, because its purpose was to enable the development of systems that were *orders of magnitude* bigger than the current status quo, with *orders of magnitude* less effort than traditional programming methodologies. As a first step in this direction, we investigated the use of a declarative language for implementing scheduling policies in Hadoop. The Hadoop scheduler assigns work to system components based on some policy (e.g., First-Come-First-Served). In Chapter 9, we specify Hadoop scheduling policies in Overlog and evaluate the resulting code through informal metrics — lines of code and development time. As we have already witnessed in previous chapters, Overlog has its own associated complexities, some of which we have addressed in a new implementation of the language called JOL (Java Overlog Library): described in Chapter 9.

The remainder of this thesis is organized as follows. Chapter 8 provides an overview of MapReduce and its open source implementation Hadoop. We focus here on the Hadoop scheduling component and batch-oriented processing dataflow implemented by Hadoop version 18.2. Readers familiar with these topics can skip to Chapter 9, where we describe BOOM-MR — an API-compliant reimplementation of the Hadoop MapReduce scheduler written in the Overlog declarative language. The resulting declarative scheduler models the (basic) First-Come-First-Served (a.k.a., FIFO) Hadoop scheduling policy in a few dozen lines of code, which took a few weeks to implement. We extended this baseline Hadoop policy with the LATE speculation

policy, by adding a mere five extra rules (12 lines of code) to our FIFO policy, which required a few days of development time. In Chapter 10, we present a pipelined version of the Hadoop MapReduce engine, where *map* and *reduce* operators no longer need to complete before emitting output data. This extension to the MapReduce model brings with it new scheduling requirements that we addressed in our declarative scheduler implementation.

Chapter 8

Hadoop MapReduce: Background

In this chapter, we review the MapReduce programming model [28] and the Hadoop system [71] — an open-source software framework that supports data-intensive distributed applications. We begin in Chapter 8.1 with the MapReduce programming model, which is based on two operations: *map* and *reduce*. Chapter 8.2 discusses the Hadoop implementation, which is comprised of a MapReduce dataflow engine, inspired by Google's MapReduce [28], and a distributed file system that models the Google File System (GFS) [33]. Chapter 8.3 summarizes the remaining chapters of this thesis as it pertains to the background material described here.

8.1 MapReduce Programming Model

MapReduce programmers expresses their computations as a series of *jobs* that process collections of data in the form of key-value pairs . Each job consists of two stages: first, a user-defined `map` function is applied to each input record to produce a list of intermediate key-value pairs. Second, a user-defined `reduce` function is called on each distinct key and list of associated values from the map output, and returns a list of output values. The MapReduce framework automatically parallelizes the execution of these functions and ensures fault tolerance.

Optionally, the user can supply a `combiner` function [28], which will be applied to the intermediate results between the map and reduce steps. Combiners are similar to reduce functions, except that they are not passed *all* the values for a given key: instead, a combiner emits an output value that summarizes the input values it was passed. Combiners are typically used to perform map-side "pre-aggregation," which reduces the amount of network traffic required between the map and reduce steps.

```
public interface Mapper<K1, V1, K2, V2> {

  void map(K1 key, V1 value, OutputCollector<K2, V2> output);

  void close();
}
```

Figure 8.1: Map function interface (Hadoop version 18.2).

8.2 Hadoop Architecture

Hadoop is composed of *Hadoop MapReduce*, an implementation of MapReduce designed for large clusters, and the *Hadoop Distributed File System* (HDFS), a file system optimized for batch-oriented workloads such as MapReduce. In most Hadoop jobs, HDFS is used to store both the input to the map step and the output of the reduce step. Note that HDFS is *not* used to store intermediate results (e.g., the output of the map step): these are kept on each node's local file system.

An Hadoop installation consists of a single master node and many worker nodes. The master, called the *JobTracker*, is responsible for accepting jobs from clients, dividing those jobs into *tasks*, and assigning those tasks to be executed by worker nodes. Each worker runs a *TaskTracker* process that manages the execution of the tasks currently assigned to that node. Each TaskTracker has a fixed number of slots for executing tasks (two maps and two reduces by default). A heartbeat protocol between each TaskTracker and the JobTracker is used to update the JobTracker's bookkeeping of the state of running tasks, and drive the scheduling of new tasks: if the JobTracker identifies free TaskTracker slots, it will schedule further tasks on the TaskTracker.

8.2.1 Map Task Execution

Each map task is assigned a portion of the input file called a *split*. By default, a split contains a single HDFS block (64MB by default), so the total number of file blocks determines the number of map tasks.

The execution of a map task is divided into two phases.

1. The *map* phase reads the task's split from HDFS, parses it into records (key/-value pairs), and applies the map function to each record.

2. After the map function has been applied to each input record, a *commit* phase registers the final output with the TaskTracker, which then informs the JobTracker that the task has finished executing.

Figure 8.1 contains the interface that must be implemented by user-defined map functions. After the `map` function has been applied to each record in the split, the

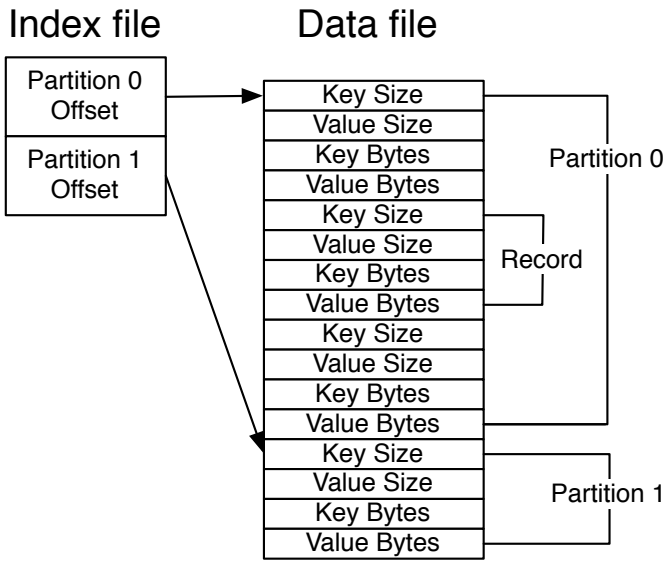


Figure 8.2: Map task index and data file format (2 partition/reduce case).

`close` method is invoked. The third argument to the `map` method specifies an *OutputCollector* instance, which accumulates the output records produced by the map function. The output of the map step is consumed by the reduce step, so the *OutputCollector* stores map output in a format that is easy for reduce tasks to consume. Intermediate keys are assigned to reducers by applying a partitioning function, so the *OutputCollector* applies that function to each key produced by the map function, and stores each record and partition number in an in-memory buffer. The *OutputCollector* spills this buffer to disk when it reaches capacity.

A spill of the in-memory buffer involves first sorting the records in the buffer by partition number and then by key. The buffer content is written to the local file system as an index file and a data file (Figure 8.2). The index file points to the offset of each partition in the data file. The data file contains only the records, which are sorted by the key within each partition segment.

During the *commit* phase, the final output of the map task is generated by merging all the spill files produced by this task into a single pair of data and index files. These files are registered with the TaskTracker before the task completes. The TaskTracker will read these files when servicing requests from reduce tasks.

8.2.2 Reduce Task Execution

The execution of a reduce task is divided into three phases.

1. The *shuffle* phase fetches the reduce task's input data. Each reduce task is assigned a partition of the key range produced by the map step, so the reduce task must fetch the content of this partition from every map task's output.

```
public interface Reducer<K2, V2, K3, V3> {

  void reduce(K2 key, Iterator<V2> values, OutputCollector<K3, V3> output);

  void close();
}
```

Figure 8.3: Reduce function interface (Hadoop version 18.2).

2. The *sort* phase groups records with the same key together.

3. The *reduce* phase applies the user-defined reduce function to each key and corresponding list of values.

In the *shuffle* phase, a reduce task fetches data from each map task by issuing HTTP requests to a configurable number of TaskTrackers at once (5 by default). The JobTracker relays the location of every TaskTracker that hosts map output to every TaskTracker that is executing a reduce task. Note that a reduce task cannot fetch the output of a map task until the map has committed its final output to disk.

After receiving its partition from all map outputs, the reduce task enters the *sort* phase. [1] The map output for each partition is already sorted by the reduce key. Therefore, the reduce task simply merges these runs together to produce a single run that is sorted by key. The task then enters the *reduce* phase, during which it invokes the user-defined reduce function for each distinct key (in sorted order) and associated list of values. The output of the reduce function is written to a temporary location on HDFS. After the reduce function has been applied to each key in the reduce task's partition, the task's HDFS output file is atomically renamed from its temporary location to its final location.

In this design, the output of both map and reduce tasks is written to disk before it can be consumed. This is particularly expensive for reduce tasks, because their output is written to HDFS. Output materialization simplifies fault tolerance, because it reduces the amount of state that must be restored to consistency after a node failure. If any task (either map or reduce) fails, the JobTracker simply schedules a new task to perform the same work as the failed task. Since a task never exports any data other than its final answer, no further recovery steps are needed.

8.3 Summary

The MapReduce interface is a good example of capturing the minimum essentials of an abstraction, making it easy to build many higher-order constructs (e.g., data analysis [70], SQL [91], machine learning [31]) while allowing significant flexibility in the system implementation. Fault-tolerance was an early part of the MapReduce

[1] Some pre-sorting work is done during the shuffle phase.

system design, and one of its most attractive features. The fault-tolerance model is predicated on the batch-oriented nature of MapReduce, allowing the recovery of a task to simply be restarting it on some (possibly alternative) node. Since no state, in the form of output data, is allowed to exit an unfinished task (map or reduce), no further recovery actions are required.

Optimization at the MapReduce level often comes in the form of scheduling policies that primarily focus on job response time. The runtime of a MapReduce job is determined by its slowest tasks. The slowest map task determines the finishing time of the *shuffle* phase since reduce tasks are not able to enter the *reduce* phase until they have received all the map outputs that belong to them. The slowest reduce task determines the finishing time of the overall job since a job does not complete until all reduce tasks complete. Speculation is a response-time optimization that executes clones of tasks deemed to be slow. Alternative speculation policies for identifying and speculatively scheduling these "straggler" tasks exist [28, 104], but there is no consensus on a policy that works well for all jobs and cluster configurations.

In Chapter 9, we describe an implementation of the Hadoop MapReduce engine in the Overlog language. Using our declarative version of Hadoop, we implemented alternative scheduling policies in Overlog that closely resemble the (policy) pseudo-code descriptions. In Chapter 10, we move from a batch-oriented execution model to a pipelined model where tasks incrementally send their output. Pipelining enables two new features in the context of MapReduce: online aggregation [45] and continuous queries. We show that a pipelined implementation of MapReduce does not sacrifice the original system interface or its ability to tolerate faults. A pipelined MapReduce model adds scheduling alternatives that we explored through policies written in Overlog.

Chapter 9

Declarative Scheduling

The Berkeley Orders Of Magnitude (BOOM) project began with an experiment in construction, by implementing a substantial piece of distributed software in a data-centric, declarative style. Upon review of recent literature on data center infrastructure (e.g., [19, 33, 29, 28]), we observed that most of the complexity in these systems were related to the management of various forms of asynchronously-updated state, including sessions, protocols and storage. Although quite complex, few of these systems involved intricate, uninterrupted sequences of computational steps. Hence, we suspected that data center infrastructure might be a good initial litmus test for our hypotheses about building distributed software.

We evaluated this hypotheses in *BOOM Analytics*: an API-compliant reimplementation of the HDFS distributed file system and the Hadoop MapReduce engine [7]. Our declarative versions of these two components were named *BOOM-FS* and *BOOM-MR*, respectively. In writing BOOM Analytics, we preserved the Java API "skin" of HDFS and Hadoop, but replaced complex internal state with relations, and implemented key system logic with code written in a declarative language. In this thesis, we focus on declarative scheduling (BOOM-MR), rather than BOOM-FS which was led by other members of the BOOM team. However, we do include some BOOM-FS results — showing its performance is on par with HDFS — to validate the JOL implementation, which was a project within this thesis.

The remainder of this chapter is organized as follows. Chapter 9.1 describes a new Java-based Overlog library, which we used to execute Overlog programs within the (Java-based) Hadoop infrastructure. In Chapter 9.2, we discuss the BOOM-MR scheduling harness; embedded in the JobTracker component of Hadoop. Chapter 9.2.1 reviews the scheduling state and protocol implemented by Hadoop version 18.2, which we modeled in our declarative code. Chapter 9.2.2 captures the entities and relationships of the Hadoop scheduler in four (catalog) tables. Using these tables, we describe a scheduling policy in Chapter 9.2.3 that models the Hadoop FIFO policy. We then extend these rules in Chapter 9.2.4 with the LATE policy for scheduling "speculative" tasks. Chapter 9.3 evaluates the performance of jobs scheduled by our declarative FIFO policy against those scheduled by the original (unmodified) Hadoop scheduler.

Finally, Chapter 9.4 examines some of the related work and Chapter 9.5 concludes with a summary of our experience with BOOM Analytics.

9.1 Java Overlog Library (JOL)

In previous chapters we witnessed P2's lack of support for stratified Datalog forced us to implement a number of imperative hacks, which often involved (event) manipulations of the underlying dataflow fixpoints. Most of these hacks were required for detecting the termination of a group of rules, which would have been implicitly handled by imposing a natural stratum boundary (e.g., count aggregate). Our workaround involved adding a number of conditions that detected stratum boundaries, and ensured that these "conditions" were evaluated in separate P2 dataflow fixpoints. This was a hard lesson, which led us to develop an entirely new Overlog implementation that supported stratified Datalog. We briefly describe this new Java Overlog Library (JOL), which we used to implement the remaining Overlog programs described in this thesis.

Like P2, JOL compiled Overlog programs into pipelined dataflow graphs of operators (similar to "elements" in the Click modular router [53]). JOL provided *metaprogramming* support akin to P2's Evita Raced extension (Chapter 3): each Overlog program is compiled into a representation that is captured in rows of tables. Program testing, optimization and rewriting could be written concisely as metaprograms in Overlog that manipulated those tables.

The JOL system matured when we targeted the Hadoop stack, which required tight integration between Overlog and Java code. The latest version of JOL included Java-based extensibility in the model of Postgres [90]. It supported Java classes as abstract data types, allowing Java objects to be stored in fields of tuples, and Java methods to be invoked on those fields from Overlog. JOL also allowed Java-based aggregation functions to run on sets of column values, and supported Java *table functions*: Java iterators producing tuples, which can be referenced in Overlog rules as ordinary relations. We made significant use of these features in BOOM Analytics; using native Hadoop data structures as column types (Chapter 9.2.2), and integrating with legacy Hadoop code (Chapters 9.2.3 and 10.4.1).

9.2 BOOM-MR: MapReduce Scheduler

In this section, we describe our declarative version of the Hadoop MapReduce scheduler, which we called BOOM-MR. Using BOOM-MR, we explored embedding a data-centric rewrite of a non-trivial component into an existing procedural system. MapReduce scheduling policies are one issue that has been treated in recent literature (e.g., [104, 103]). To enable credible work on MapReduce scheduling, we wanted to remain true to the basic structure of the Hadoop MapReduce codebase, so we

Name	*Description*	*Relevant attributes*
job	Job definitions	JobId, Priority, SubmitTime, Status, JobConf
task	Task definitions	JobId, TaskId, Type, Partition, Status
taskAttempt	Task instance	JobId, TaskId, AttemptId, Progress, State, Phase, Tracker, InputLoc, Start, Finish
taskTracker	TaskTracker state	Name, Hostname, State, MapCount, ReduceCount, MaxMap, MaxReduce

Table 9.1: BOOM-MR relations defining JobTracker state.

proceeded by understanding that code, mapping its core state into a relational representation, and then writing Overlog rules to manage that state in the face of new messages delivered by the existing Java APIs.

9.2.1 Hadoop MapReduce Scheduler

We briefly review the Hadoop scheduling logic that we modeled in Overlog. The Hadoop architecture consists of a single master node called the *JobTracker* that manages a number of worker nodes called *TaskTrackers*. A job is divided into a set of map and reduce *tasks*. The JobTracker assigns tasks to worker nodes. Each map task reads an input chunk from the distributed file system, runs a user-defined map function, and partitions output key/value pairs into hash buckets on the local disk. Reduce tasks are created for each hash bucket. Each reduce task fetches the corresponding hash buckets from all mappers, sorts locally by key, runs a user-defined reduce function and writes the results to the distributed file system.

Each TaskTracker has a fixed number of slots for executing tasks (two maps and two reduces by default). A heartbeat protocol between each TaskTracker and the JobTracker is used to update the JobTracker's bookkeeping of the state of running tasks, and drive the scheduling of new tasks: if the JobTracker identifies free TaskTracker slots, it will schedule further tasks on the TaskTracker. Also, Hadoop will attempt to schedule *speculative* tasks to reduce a job's response time if it detects "straggler" nodes [28].

9.2.2 Table-izing MapReduce

BOOM-MR is a port of the Hadoop JobTracker code to Overlog. Here, we identify the key state maintained by the JobTracker. This includes both data structures to track the ongoing status of the system and transient state in the form of messages sent and received by the JobTracker. We captured this information in the four Overlog relations shown in Table 9.1.

The `job` relation contains a single row for each job submitted to the JobTracker.

In addition to some basic metadata, each job tuple contains an attribute called the *JobConf*, which holds a Java object constructed by legacy Hadoop code. This object captures the configuration parameters that pertain to a single MapReduce job. The `task` relation identifies each task within a job using attributes that specify the task type (map or reduce), the input "partition" (a chunk for map tasks, a bucket for reduce tasks), and the current running status.

A task may be attempted more than once, due to speculation or if the initial execution attempt failed. The `taskAttempt` relation maintains the state of each such attempt (one per row). In addition to a progress percentage and a state (running/-completed), we maintain a task phase i.e., reduce tasks can be in any one of three phases: copy, sort, or reduce. The *Tracker* attribute identifies the TaskTracker assigned to execute the task attempt. Map tasks also need a record containing the location of their input data, which is given by *InputLoc*.

The `taskTracker` relation identifies each TaskTracker in the cluster with a unique name. This relation includes attributes that provide the hostname, current running state, and the TaskTracker workload. Specifically, the *MapCount* and *ReduceCount* attributes specify the current number of map and reduce tasks that are executing on the TaskTracker. The maximum number of map and reduce tasks that the TaskTracker is able to support is given by the *MaxMap* and *MaxReduce* attributes; this is in keeping with the Hadoop implementation, which specifies a fixed number of slots that can execute tasks.

9.2.3 MapReduce Scheduling in Overlog

MapReduce scheduling has been the subject of much recent research [103, 104, 6, 17, 87, 40], and one of our early motivations for building BOOM Analytics was to make this research extremely easy to carry out. In our initial BOOM-MR prototype, we implemented Hadoop's default First-Come-First-Served (or FIFO) policy for task scheduling, which we captured in 9 rules (96 lines). We then extended these rules with the recently-proposed LATE policy [104] to evaluate both (a) the difficulty of prototyping a new policy, and (b) the faithfulness of our Overlog-based execution to that of Hadoop using two separate speculation algorithms.

First-Come-First-Served Scheduling

The FIFO policy schedules tasks from the job with the highest priority. A job's scheduling order is defined by its *Priority* followed by its *SubmitTime* (see `job` schema in Table 9.1). The tasks from the job that is first in the scheduling order are scheduled before the tasks in any other jobs.

Figure 9.1 captures this constraint in three rules, which identify the job whose tasks are considered first when TaskTracker slots are available. Rule `s1` identifies the job with the overall minimum priority, while rule `s2` determines, for each job

```
s1 minWaitingJobPriority(a_min<Priority>) :-
   job(JobId, Priority, Status, ...),
   Status < JobStatus.FINISHED;

s2 minWaitingJobPrioritySubmitTime(Priority, a_min<SubmitTime>) :-
   job(JobId, Priority, Status, SubmitTime, ...),
   Status < JobStatus.FINISHED;

s3 highestPriorityJob(JobId) :-
   minWaitingJobPriority(Priority),
   minWaitingJobPrioritySubmitTime(Priority, SubmitTime),
   job(JobId, Priority, Status, SubmitTime, ...);
```

Figure 9.1: The highest priority job that still has unscheduled tasks ($StartTime < 0$).

priority, what is the earliest submit time. Both rules `s1` and `s2` only consider jobs that have unscheduled tasks, shown here by considering the `Status < JobStatus.FINISHED` predicate. Rule `s3` joins the result of rules `s1` and `s2` to identify the overall highest priority job with unscheduled tasks. The `highestPriorityJob` predicate is used to constrain task scheduling rules to only consider unscheduled tasks from the specified job.

Scheduling individual tasks from the highest priority job occurs when a TaskTracker performs a heartbeat exchange with the JobTracker and has some number of available map or reduce task slots. Tasks are scheduled based on slot availability; if a task slot is available then schedule a task from the job with the highest priority. To avoid data movement costs, the scheduling policy tries to schedule the map task close to a machine that hosts its input data. Ideally, it schedules a map task whose input resides on the same machine or rack. If no such option exists then an arbitrary map task is scheduled, without considering other queued jobs. Concurrent to this work, Zaharia et al. introduced Delay Scheduling [103], which delayed scheduling tasks on machines that did not offer good locality. Their results achieved perfect locality — all tasks scheduled close to the input data — and no task was delayed for more than five seconds.

Returning to the default Hadoop policy, Figure 9.2 shows two rules that together implement, a locality aware, Hadoop FIFO policy. When a TaskTracker heartbeat is received, rule `s4` assigns a locality metric to unscheduled tasks that belong to the highest priority job. JOL supports the ability to add Java code at the end of a rule body, delineated within brackets { ... }. This Java code executes last in the rule body, and will only see those tuples that represent actual deductions. [1] In rule `s4`, the bracketed Java code assigns a *locality* metric according to the proximity of the heartbeat TaskTracker to the map input data.

The result of rule `s4` is evaluated in rule `s5`, which schedules the map tasks whose input resides closest to the heartbeat TaskTracker. The **bottomK** aggregate orders

[1] A useful feature for `printf` style debugging.

```
/* Assign each task a locality score on the given tracker. */
s4 mapTaskLocality(TaskId, Tracker, Locality) :-
   heartbeat(Tracker, TrackerStatus, MapSlots, ReduceSlots),
   hightestPriorityJob(JobId),
   task(JobId, TaskId, Type, _, InputSplits, StartTime, _),
   StartTime < 0, Type == ''map'',
   {
     if (InputSplits.contains(TrackerStatus.getHost())) {
       Locality := 1; // same machine
     } else if (InputSplits.contains(TrackerStatus.getRack()) {
       Locality := 2; // same rack
     } else {
       Locality := 3;
     }
   };

/* For each task tracker, list the k best map tasks to
   schedule, where k == MapSlots. The result of this
   will be added to the schedule relation, which is
   returned to the TaskTracker. */
s5 schedule(Tracker, bottomK<MapID, MapSlots>) :-
   mapTaskLocality(TaskId, Tracker, Locality),
   heartbeat(Tracker, TrackerStatus, MapSlots, ReduceSlots),
   TrackerStatus == TaskTrackerStatus.RUNNING,
   MapSlots > 0,
   MapID := new OrderedMapID(TaskId, Locality);
```

Figure 9.2: Map task locality priority scheduler.

the $MapID$s from lowest to highest $Locality$ and chooses the lowest K map tasks in this order, not exceeding the number of available map slots ($MapSlots$). Each result tuple from rule `s5` is converted, through a few imperative steps in the Java language, into a schedule action message that is returned to the TaskTracker in the RPC call made to the JobTracker. The reduce task scheduling rule simply schedules reduces tasks from the highest priority job based on the availability of reduce slots from the heartbeat TaskTracker, as per stock Hadoop.

9.2.4 Task Speculation in Overlog

With the basic scheduling logic behind us, we turn now to the topic of scheduling speculative tasks. The LATE policy presents a scheme for scheduling speculative tasks based on *straggler* tasks [104]. There are two aspects to each policy: choosing which tasks to speculatively re-execute, and choosing TaskTrackers to run those tasks. Original Hadoop re-executes a task if its progress is more than 0.2 (on a scale of [0..1]) below the mean progress of similar tasks. LATE, on the other hand, chooses to re-execute tasks via an *estimated finish time* metric that is based on the task's *progress rate*. Moreover, it avoids assigning speculative tasks to TaskTrackers that exhibit slow performance executing similar tasks, in hopes of preventing further stragglers.

The LATE policy is specified in the paper [104] via three lines of pseudocode, which makes use of three performance related statistics called $SlowNodeThreshold$, $SlowTaskThreshold$ and $SpeculativeCap$. The first two statistics correspond to the 25^{th} percentiles of progress rates across TaskTrackers and across tasks, respectively. The $SpeculativeCap$ indicates the maximum number of speculative tasks allowed at any given time, which is suggested to be set at 10% of the total available task slots.

We compute these thresholds via the five Overlog rules shown in Figure 9.3. A task is only considered for speculation if its progress rate falls below the $SlowTaskThreshold$ in its given category: job identifier ($JobID$) and task type ($Type$). Queries `l1` and `l2` maintain this threshold value for each category. Query `l1` determines the progress rate for a given task based on its current progress and running time. Query `l2` computes the $SlowTaskThreshold$, for each category, by determining the lower 25^{th} percentile of the progress rates.

The LATE policy ensures that speculative tasks execute on "fast" nodes by pruning TaskTracker nodes whose rate of progress for a given task category fall below some threshold. Queries `l3` and `l4` maintain this threshold value for each category. The first query `l3`, computes the average progress that a given TaskTracker has made for each task category and stores that result in the `trackerPR` table. Query `l4` computes the $SlowNodeThreshold$ for each category by determining the 25th percentile for each category of progress rates stored in the `trackerPR` table. Finally, query `l5` counts the number of slots that can be used for task speculation. Integrating the rules into BOOM-MR required two additional Overlog rules that 1) identify tasks to speculatively re-execute, and 2) select an ideal TaskTracker(s) on which to execute those tasks, all while obeying the SpeculativeCap value.

```
/* Compute progress rate per task */
l1 taskPR(JobId, TaskId, Type, ProgressRate) :-
   task(JobId, TaskId, Type, _, _, _, Status),
   Status.state() == RUNNING,
   Time := Status.finish() > 0 ? Status.finish() :
             java.lang.System.currentTimeMillis(),
   ProgressRate := Status.progress() / (Time - Status.start());

/* For each job, compute 25th pctile rate across tasks */
l2 slowTaskThreshold(JobId, Type, a_percentile<0.25, PRate>) :-
   taskPR(JobId, TaskId, Type, PRate);

/* Compute progress rate per tracker */
l3 trackerPR(Tracker, JobId, Type, a_avg<PRate>) :-
   task(JobId, TaskId, Type, _),
   taskAttempt(JobId, TaskId, _, Progress, State, Phase,
               Tracker, Start, Finish),
   State != FAILED,
   Time := Finish > 0 ? Finish : java.lang.System.currentTimeMillis(),
   PRate := Progress / (Time - Start);

/* For each job, compute 25th pctile rate across all trackers */
l4 slowNodeThreshold(JobId, Type, a_percentile<0.25, AvgPRate>) :-
   trackerPR(_, JobId, Type, AvgPRate);

/* Compute available map/reduce slots that can be used for
   speculation. */
l5 speculativeCap(a_sum<MapSlots>, a_sum<ReduceSlots>) :-
   taskTracker( ... MapCount, ReduceCount, MaxMap, MaxReduce),
   MapSlots    := java.lang.Math.ceil(0.1 * (MaxMap - MapCount)),
   ReduceSlots := java.lang.Math.ceil(0.1 * (MaxReduce - ReduceCount));
```

Figure 9.3: Overlog to compute statistics for LATE.

9.3 Evaluation

We now validate our declarative specification of both Hadoop's default FIFO policy and the LATE policy proposed by Zaharia et al. [104]. Our goals were both to evaluate the difficulty of building a new policy, and to confirm the faithfulness of our Overlog-based JobTracker to the Hadoop JobTracker when using a logically identical scheduling policy and with the additional LATE policy.

We evaluated our Overlog policies using a 101-node virtual cluster on Amazon EC2. One node executed the Hadoop JobTracker and the HDFS NameNode, while the remaining 100 nodes served as "workers" for running the Hadoop TaskTrackers and HDFS DataNodes. Each TaskTracker was configured to support up to two map tasks and two reduce tasks simultaneously. The master node ran on a "high-CPU extra large" EC2 instance with 7.2 GB of memory and 8 virtual cores. Our worker nodes executed on "high-CPU medium" EC2 instances with 1.7 GB of memory and 2 virtual cores. Each virtual core is the equivalent of a 2007-era 2.5Ghz Intel Xeon processor.

9.3.1 FIFO policy

While improved performance was not a goal of our work, we wanted to ensure that the performance of BOOM Analytics was competitive with Hadoop. The workload was a wordcount job on a 30 GB file, using 481 map tasks and 100 reduce tasks.

Figure 9.4 contains four graphs comparing the performance of different combinations of Hadoop MapReduce, HDFS, BOOM-MR, and BOOM-FS. Each graph reports a cumulative distribution of the elapsed time in seconds from job startup to map or reduce task completion. The map tasks complete in three distinct "waves." This is because only 2×100 map tasks can be scheduled at once. Although all 100 reduce tasks can be scheduled immediately, no reduce task can finish until all maps have been completed because each reduce task requires the output of all map tasks.

The lower-left graph describes the performance of Hadoop running on top of HDFS, and hence serves as a baseline for the subsequent graphs. The upper-left graph details BOOM-MR running over HDFS. This graph shows that map and reduce task durations under BOOM-MR are nearly identical to Hadoop 18.2. The lower-right and upper-right graphs detail the performance of Hadoop MapReduce and BOOM-MR running on top of BOOM-FS, respectively. BOOM-FS performance is slightly slower than HDFS, but remains competitive.

9.3.2 LATE policy

We now compare the behavior of our LATE implementation with the results observed by Zaharia et al. using Hadoop MapReduce. LATE focuses on how to improve job completion time by reducing the impact of "straggler" tasks. To simulate stragglers,

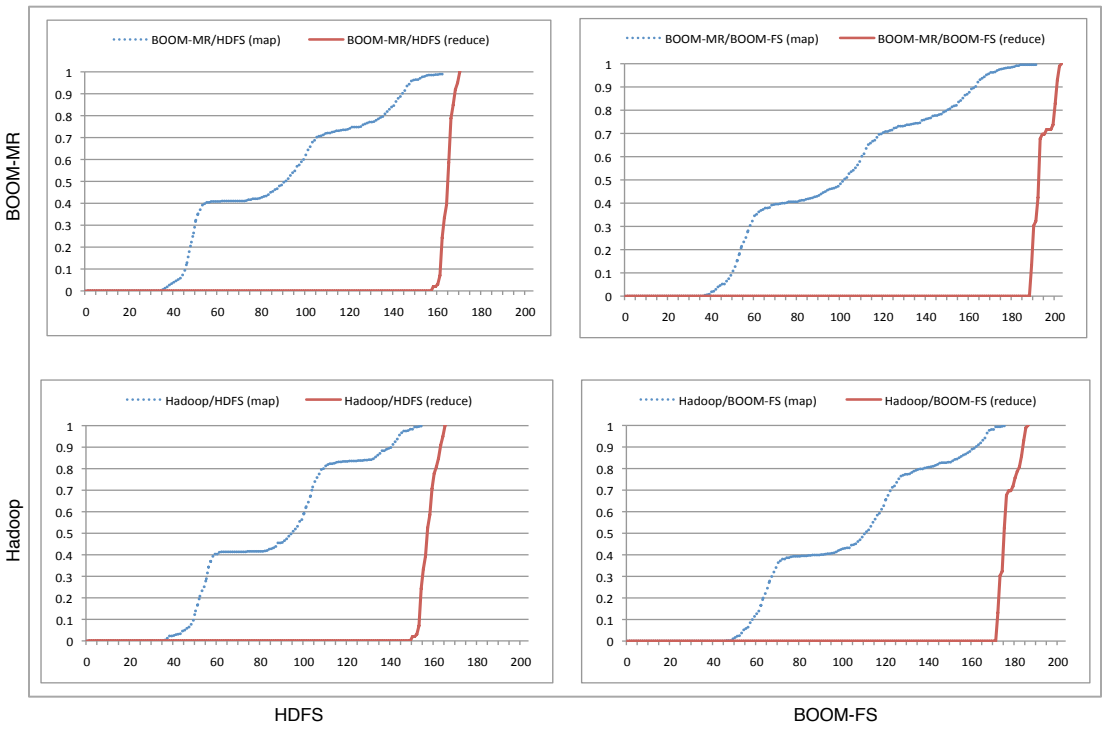


Figure 9.4: CDFs representing the elapsed time between job startup and task completion for both map and reduce tasks, for all combinations of Hadoop and BOOM-MR over HDFS and BOOM-FS. In each graph, the horizontal axis is elapsed time in seconds, and the vertical represents the percentage of tasks completed.

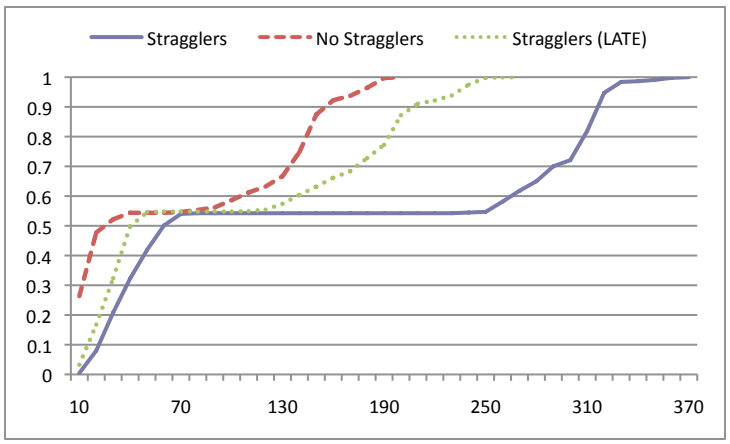


Figure 9.5: CDF of reduce task duration (secs), with and without stragglers.

we artificially placed additional load on six nodes. We ran the same wordcount job on 30 GB of data; using 481 map tasks and 400 reduce tasks, which produced two distinct "waves" of reduce tasks. We ran each experiment five times, and report the average over these runs.

Figure 9.5 shows the reduce task duration CDF for three different configurations. The plot labeled "No Stragglers" represents normal load, while the "Stragglers" and "Stragglers (LATE)" plots describe performance in the presence in stragglers using the default FCFS policy and the LATE policy, respectively. We omit map task durations, because adding artificial load had little effect on map task execution — it just resulted in slightly slower growth from just below 100% to completion.

The 200 reduce tasks were scheduled concurrently with the map step. This first wave of reduce tasks cannot enter the reduce phase until all the map tasks have completed, which increased their duration, and resulted in the large runtime durations indicated in the right portion of the graph. The second wave of 200 reduce tasks did not experience this delay due to unfinished map work since these reduce tasks were scheduled after all map tasks had finished. The second wave of reduce tasks are reported in the left portion of the graph. Consequently, stragglers had less of an impact on the second wave of reduce tasks since fewer resources (i.e., no map work) were being consumed. Figure 9.5 shows this effect, and also demonstrates how the LATE implementation in BOOM Analytics handles stragglers much more effectively than the default Hadoop policy. This echoes the results reported by Zaharia et al. [104]

9.4 Related Work

Declarative and data-centric languages have traditionally been considered useful in very few domains, but things have changed substantially in recent years. MapReduce [28] has popularized functional dataflow programming with new audiences in

computing. Also, a surprising breadth of recent research projects have proposed and prototyped declarative languages, including overlay networks [63], three-tier web services [101], natural language processing [30], modular robotics [12], video games [97], file system metadata analysis [41], and compiler analysis [55].

Most of the languages cited above are declarative in the same sense as SQL: they are based in first-order logic. Some — notably MapReduce, but also SGL [97] — are algebraic or dataflow languages, used to describe the composition of operators that produce and consume sets or streams of data. Although arguably imperative, they are far closer to logic languages than to traditional imperative languages like Java or C, and are often amenable to set-oriented optimization techniques developed for declarative languages [97]. Declarative and dataflow languages can also share the same runtime, as demonstrated by recent integrations of MapReduce and SQL in Hive [91], DryadLINQ [102], HadoopDB [6], and products from vendors such as Greenplum and Aster.

Concurrent with our work, the Erlang language was used to implement a simple MapReduce framework called Disco [68] and a transactional DHT called Scalaris with Paxos support [81]. Philosophically, Erlang revolves around concurrent *actors*, rather than data. A closer comparison of actor-oriented and data-centric design styles is beyond the scope of this dissertation, but an interesting topic for future work.

9.5 Summary

The initial version of BOOM-MR required one person-month of development time and an additional two person-months debugging and tuning BOOM-MR's performance for large jobs. The final version of BOOM-MR contained declarative specifications for the core task scheduler (9 rules), the speculative task scheduler (5 rules), recovery from failed tasks (3 rules), and maintenance of various job and task related statistics (5 rules). In total, BOOM-MR consisted of 22 Overlog rules in 156 lines of code, and 1269 lines of Java. BOOM-MR was based on Hadoop version 18.2; we estimate that we removed 6,573 lines of code (out of 88,863) from the `org.apache.hadoop.mapred` Hadoop package.

In the end, we found that scheduling policies were a good fit for a declarative language like Overlog. In retrospect, this is because scheduling can be decomposed into two tasks: *monitoring* system state and applying *policies* for how to react to state changes. Monitoring is well-handled by Overlog: we found that the statistics about TaskTracker state required by the LATE policy are naturally realized as aggregate functions, and JOL took care of automatically updating those statistics as new messages from TaskTrackers arrived. In the next chapter, we will look at importing statistics taken from the output of a MapReduce job that is continuously monitoring machine and process level statistics. Once these near real-time monitoring statistics have been imported into JOL, we can build some very interesting scheduling policies around them.

It is also unsurprisingly that a logic language should be well-suited to specifying policy. We found the BOOM-MR scheduler much simpler to extend and modify than the original Hadoop Java code, as demonstrated by our experience with LATE. Informally, the Overlog code in BOOM-MR seems about as complex as it should be: Hadoop's MapReduce task coordination logic is a simple and clean design, and the compactness of BOOM-MR reflects that simplicity appropriately. The extensibility of BOOM-MR benefited us when we extended the MapReduce batch-oriented model to one that pipelined data between operators (Chapter 10); supporting both online aggregation [45] and stream processing [67] jobs.

Chapter 10

MapReduce Online

MapReduce is typically applied to large batch-oriented computations that do not require any real-time completion constraints. The Google MapReduce framework [28] and open-source Hadoop system reinforce this usage model through a batch-processing implementation strategy: the entire output of each map and reduce task is *materialized* to a local file before it can be consumed by the next stage. Materialization allows for a simple and elegant checkpoint/restart fault-tolerance mechanism that is critical in large deployments, which have a high probability of slowdowns or failures at worker nodes. However, batch-processing is not a requirement for fault-tolerance. Moreover, batch-processing prevents many online data processing strategies [45, 4, 67, 22] and its aggressive materialization strategy can be costly in terms of efficiency e.g., energy [23].

In this chapter, we propose an alternative MapReduce architecture in which intermediate data is *pipelined* between operators, while preserving the programming interfaces and fault-tolerance properties of previous MapReduce frameworks. To validate our design, we developed the Hadoop Online Prototype (HOP): a pipelined version of Hadoop.[1]

Pipelining provides several important advantages to a MapReduce framework, but also raises new design challenges. We highlight the potential benefits first:

- Since reducers begin processing data as soon as it is produced by mappers, they can generate and refine an approximation of their final answer during the course of execution. This technique, known as *online aggregation* [45], can provide initial estimates of results several orders of magnitude faster than the final result. We describe how we adapted online aggregation to our pipelined MapReduce architecture in Chapter 10.2.

- Pipelining widens the domain of problems to which MapReduce can be applied. In Chapter 10.3, we show how HOP can be used to support *continuous queries*: MapReduce jobs that run continuously, accepting new data as it arrives and

[1]The source code for HOP can be downloaded from `http://code.google.com/p/hop/`

analyzing it immediately. This allows MapReduce to be used for applications such as event monitoring and stream processing.

- Pipelining delivers data to downstream operators more promptly, which can increase opportunities for parallelism, improve utilization, and reduce response time. A thorough performance study is a topic for future work; however, in Chapter 10.1.4 we present some initial performance results which demonstrate that pipelining can reduce job completion times by up to 25% in some scenarios.

We develop the design of HOP's pipelining scheme in Chapter 10.1, keeping the focus on traditional batch processing tasks. Pipelining raises several design challenges. First, Google's attractively simple MapReduce fault-tolerance mechanism is predicated on the materialization of intermediate state. In Chapter 10.1.3, we show that fault-tolerance can coexist with pipelining, by allowing producers to periodically ship data to consumers in parallel with data materialization. A second challenge arises from the greedy communication implicit in pipelines, which is at odds with batch-oriented optimizations supported by "combiners": map-side code that reduces network utilization by performing pre-aggregation before communication. We discuss how the HOP design addresses this issue in Chapter 10.1.1. Finally, pipelining requires that producers and consumers are co-scheduled intelligently. In Chapter 10.4.1, we discuss some declarative scheduling policies that try to fill the pipeline early — by scheduling downstream operators first — and enforce a complete pipeline for continuous queries.

The remaining portions of this chapter focus on applications of HOP and scheduling policies related to those applications. In Chapter 10.2, we show how HOP can support online aggregation for long-running jobs and illustrate the potential benefits of that interface to MapReduce programmers. Chapter 10.3 describes our support for continuous MapReduce jobs over data streams and demonstrate an example of a near-real-time cluster monitoring application. In Chapter 10.4, we return to the topic of scheduling to address the new challenges raised by these HOP applications. Chapter 10.4.1 describes our port of the BOOM-MR declarative scheduler to HOP and some new Overlog scheduling policies that deal with online aggregation and continuous jobs. Chapter 10.5 introduces a new speculation policy based on statistics collected by a (continuous) MapReduce monitoring job described in Chapter 10.3.2. Finally, Chapter 10.6 concludes with some related work.

10.1 Pipelined MapReduce

We begin with a description of our Hadoop extensions that support pipelining between tasks (Chapter 10.1.1) and jobs (Chapter 10.1.2). We describe how our design supports fault-tolerance (Chapter 10.1.3) and compare the performance of HOP under both pipelining and blocking execution modes (Chapter 10.1.4).

10.1.1 Pipelining Within A Job

As described in Chapter 8.2.2, reduce tasks traditionally issue HTTP requests to *pull* their input from each TaskTracker that hosted a map task belonging to the same job. A TaskTracker is responsible for serving these HTTP requests, which could occur long after the map task's execution. This means that map task execution is completely decoupled from reduce task execution. To support pipelining, we modified the TaskTracker serving component to *push* data to reducers as it is produced by the map tasks, while still maintaining the decoupling of these two steps. To give an intuition for how this works, we begin by describing a straightforward pipelining design, and then discuss the changes we had to make to achieve good performance.

Naïve Pipelining

We begin with a naïve implementation that sends data directly from map to reduce tasks via a TCP socket. Immediately, this design couples the execution of map and reduce task executions, forcing us to schedule *all* reduce tasks before any one map task. Consequently, this design does not scale, most notably when there is not sufficient reduce task slot capacity, but there are other ramifications that we discuss here before converging on the true HOP design.

Recall, that when a client submits a new job to Hadoop, the JobTracker assigns the map and reduce tasks associated with the job to the available TaskTracker slots. For purposes of *this* discussion, we must assume that there are enough free slots to assign all reduce tasks in a job. We modified Hadoop so that each reduce task contacts every map task upon initiation of the job, and opens a TCP socket which will be used to pipeline the output of the map function. As each map output record is produced, the mapper determines which partition (reduce task) the record should be sent to, and immediately sends it via the appropriate socket.

A reduce task accepts the pipelined data it receives from each map task and stores it in an in-memory buffer, spilling sorted runs of the buffer to disk as needed. Once the reduce task learns that every map task has completed, it performs a final merge of all the sorted runs and applies the user-defined reduce function as normal.

Refinements

While the algorithm described above is straightforward, it suffers from several practical problems. First, it is possible that there will not be enough slots available to schedule every task in a new job. Opening a socket between every map and reduce task also requires a large number of TCP connections. A simple tweak to the naïve design solves both problems: if a reduce task has not yet been scheduled, any map tasks that produce records for that partition simply write them to disk. When the map task completes, it registers the output it was not able to send with the host TaskTracker serving component. Once the reduce task is assigned a slot, it can then

pull the records from the map task's host TaskTracker, as in regular Hadoop. To reduce the number of concurrent TCP connections, each reducer can be configured to pipeline data from a bounded number of mappers at once; the reducer will pull data from the remaining map tasks in the traditional Hadoop manner.

Our initial pipelining implementation suffered from a second problem: the map function was invoked by the same thread that wrote output records to the pipeline sockets. This meant that if a network I/O operation blocked (e.g., because the reducer was over-utilized), the mapper was prevented from doing useful work. Pipeline stalls should not prevent a map task from making progress – especially since, once a task has completed, it frees a TaskTracker slot to be used for other purposes. We solved this problem by running the map function in a separate thread that stores its output in an in-memory buffer, and then having another thread periodically send the contents of the buffer to the connected reducers. [2]

Granularity of Map Output

Another problem with the naïve design is that it eagerly sends each record as soon as it is produced, which prevents the use of map-side combiners. Imagine a job where the reduce key has few distinct values (e.g., gender), and the reduce applies an algebraic aggregate function (e.g., count). As discussed in Chapter 8.1, combiners allow map-side "pre-aggregation": by applying a reduce-like function to each distinct key at the mapper, network traffic can often be substantially reduced. Eagerly pipelining each record as it is produced prevents the use of these map-side combiners.

Another related problem is that eager pipelining moves some of the sorting work from the mapper to the reducer. Recall from Chapter 8.2.1, that in the blocking architecture, map tasks generate sorted spill files: all the reduce task must do is merge together the pre-sorted map output for each partition. In the naïve pipelining design, map tasks send output records as they are generated, so a reducer (scheduled early) must perform a full external sort. Because the number of map tasks typically far exceeds the number of reduces [28], moving more work to the reducer increased response time, as shown in our experiments (Chapter 10.1.4).

To avoid a heavy reduce task sort, instead of sending the buffer contents to reducers directly, we wait for the buffer to grow to a threshold size. The mapper then (quick) sorts the output by partition and reduce key, applies the combiner function, and writes the buffer to disk using the Hadoop spill file format described in Figure 8.2. Next, we arranged for the TaskTracker serving component at each node to handle pipelining data to reduce tasks. Map tasks register spill files with the TaskTracker via RPCs. [3] If the reducers are able to keep up with the production of map

[2]This code was based on the existing Hadoop SpillThread component, which is responsible for writing map output to disk concurrently with the "map function."

[3]We extended the existing RPC Hadoop interface to include information on individual spill files. Having the spill files be in the same format allowed us to reuse much of the stock Hadoop serving code i.e., I/O file formats/streams.

outputs and the network is not a bottleneck, a spill file will be sent to a reducer soon after it has been produced (in which case, the spill file is likely still resident in the map machine's kernel buffer cache). However, if a reducer begins to fall behind, the number of unsent spill files will grow.

When a map task generates a new spill file, it first queries the TaskTracker for the number of unsent spill files. If this number grows beyond a certain threshold (two unsent spill files in our experiments), the map task does not immediately register the new spill file with the TaskTracker. Instead, the mapper will accumulate multiple spill files. Once the queue of unsent spill files falls below the threshold, the map task merges and combines the accumulated spill files into a single file, and then resumes registering its output with the TaskTracker. This simple flow control mechanism has the effect of *adaptively* moving load from the reducer to the mapper or vice versa, depending on which node is the current bottleneck.

A similar mechanism is also used to control how aggressively the combiner function is applied. The map task records the ratio between the input and output data sizes whenever it invokes the combiner function. If the combiner is effective at reducing data volumes, the map task accumulates more spill files (and applies the combiner function to all of them) before registering that output with the TaskTracker for pipelining.[4]

The connection between pipelining and adaptive query processing techniques has been observed elsewhere (e.g., [13, 79]). The adaptive scheme outlined above is relatively simple, but we believe that adapting to feedback along pipelines has the potential to significantly improve the utilization of MapReduce clusters.

10.1.2 Pipelining Between Jobs

Many practical computations cannot be expressed as a single MapReduce job, and the outputs of higher-level languages like Pig [70] typically involve multiple jobs. In the traditional Hadoop architecture, the output of each job is written to HDFS in the reduce step and then immediately read back from HDFS by the map step of the next job. Furthermore, the JobTracker cannot schedule a consumer job until the producer job has completed, because scheduling a map task requires knowing the HDFS block locations of the map's input split.

In our modified version of Hadoop, the reduce tasks of one job can optionally pipeline their output directly to the map tasks of the next job, sidestepping the need for expensive fault-tolerant storage in HDFS for what amounts to a temporary file. Unfortunately, the computation of the reduce function from the previous job and the map function of the next job cannot be overlapped: the final result of the reduce step cannot be produced until all map tasks have completed, which prevents effective

[4]Our current prototype uses a simple heuristic: if the combiner reduces data volume by $\frac{1}{k}$ on average, we wait until k spill files have accumulated before registering them with the TaskTracker. A better heuristic would also account for the computational cost of applying the combiner function.

pipelining. However, we describe later how online aggregation and continuous query pipelines can publish "snapshot" outputs that can indeed pipeline between jobs.

10.1.3 Fault Tolerance

Our pipelined Hadoop implementation is robust to the failure of both map and reduce tasks. To recover from map task failures, we added bookkeeping to the reduce task to record which map task produced each pipelined spill file. To simplify fault-tolerance, the reducer treats the output of a pipelined map task as "tentative" until the JobTracker informs the reducer that the map task has committed successfully. The reducer can merge together spill files generated by the same uncommitted mapper, but will not combine those spill files with the output of other map tasks until it has been notified that the map task has committed. Thus, if a map task fails, each reduce task can ignore any tentative spill files produced by the failed map attempt. The JobTracker will take care of scheduling a new map task attempt, as in stock Hadoop.

If a reduce task fails and a new copy of the task is started, the new reduce instance must be sent all the input data that was sent to the failed reduce attempt. If map tasks operated in a purely pipelined fashion and discarded their output after sending it to a reducer, this would be difficult. Therefore, map tasks retain their output data on the local disk for the complete job duration. This allows the map's output to be reproduced if any reduce tasks fail. For batch jobs, the key advantage of our architecture is that reducers are not blocked waiting for the complete output of the task to be written to disk.

Our technique for recovering from map task failure is straightforward, but places a minor limit on the reducer's ability to merge spill files. To avoid this, we envision introducing a "checkpoint" concept: as a map task runs, it will periodically notify the JobTracker that it has reached offset x in its input split. The JobTracker will notify any connected reducers; map task output that was produced before offset x can then be merged by reducers with other map task output as normal. To avoid duplicate results, if the map task fails, the new map task attempt resumes reading its input at offset x. This technique would also reduce the amount of redundant work done after a map task failure or during speculative execution of "backup" tasks [28].

10.1.4 Performance Evaluation

A thorough performance comparison between pipelining and blocking is not the focus of this work. However, as future work we plan to investigate a rule-based (e.g., Evita Raced) optimizer for Hadoop MapReduce that considers pipelined plans in its search strategy. Here, we demonstrate that pipelining can reduce job completion times in some configurations and should be considered by any such optimizer.

We report performance using both large (512MB) and small (32MB) HDFS block

sizes using a single workload (a wordcount job over randomly-generated text). Since the words were generated using a uniform distribution, map-side combiners were ineffective for this workload. We performed all experiments using relatively small clusters of Amazon EC2 nodes. We also did not consider performance in an environment where multiple concurrent jobs are executing simultaneously.

Background and Configuration

Before diving into the performance experiments, it is important to further describe the division of labor in a HOP job, which is broken into task phases. A map task consists of two work phases: *map* and *sort*. Much of the work performed during the job happens in the *map* phase, where the map function is applied to each record in the input and subsequently sent to an output buffer. Once the entire input has been processed, the map task enters the *sort* phase, where a final merge sort of all intermediate spill files is performed before registering the final output with the TaskTracker. The progress reported by a map task corresponds to the *map* phase, which is overlapped with many in-memory record buffer sorts and subsequent spills to local files.

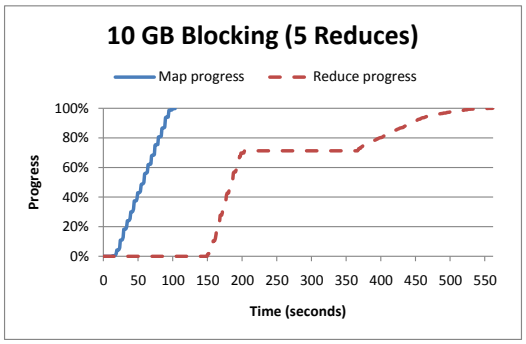
A reduce task in HOP is divided into three work phases: *shuffle*, *reduce*, and *commit*. In the *shuffle* phase, reduce tasks receive their portion of the output from each map. In HOP, the *shuffle* phase consumes 75% of the overall reduce task progress while the remaining 25% is allocated to the *reduce* and *commit* phase. [5] In the *shuffle* phase, reduce tasks periodically perform a merge sort on the already received map output. These intermediate merge sorts decrease the amount of sorting work performed at the end of the *shuffle* phase. After receiving its portion of data from all map tasks, the reduce task performs a final merge sort and enters the *reduce* phase.

By pushing work from map tasks to reduce tasks more aggressively, pipelining can enable better overlapping of map and reduce computation, especially when the node on which a reduce task is scheduled would otherwise be underutilized. However, when reduce tasks are already the bottleneck, pipelining offers fewer performance benefits, and may even hurt performance by placing additional load on the reduce nodes.

The *sort* phase in the map task minimizes the merging work that reduce tasks must perform at the end of the *shuffle* phase. When pipelining is enabled, the *sort* phase is avoided since map tasks have already sent some fraction of the spill files to concurrently running reduce tasks. Therefore, pipelining increases the merging workload placed on the reducer. The adaptive pipelining scheme described in Chapter 10.1.1 attempts to ensure that reduce tasks are not overwhelmed with additional load.

We used two Amazon EC2 clusters depending on the size of the experiment:

[5] The stock version of Hadoop divides the reduce progress evenly among the three phases. We deviated from this approach because we wanted to focus more on the progress during the *shuffle* phase.



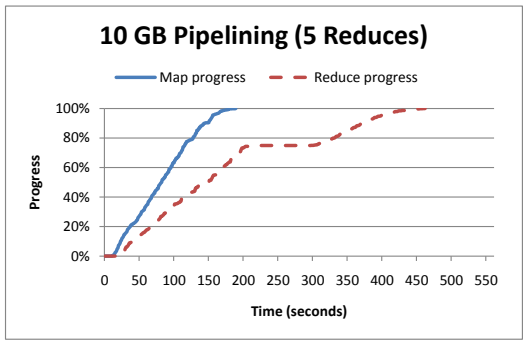
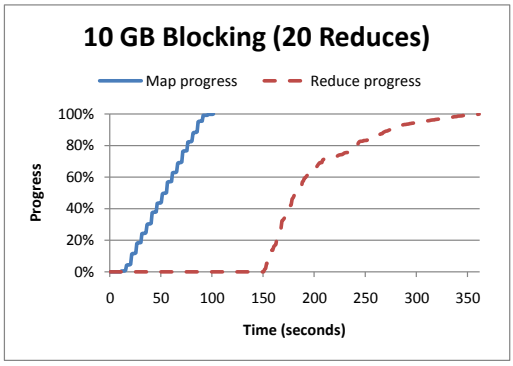


Figure 10.1: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 5 reduce tasks (512MB block size). The total job runtimes were 561 seconds for blocking and 462 seconds for pipelining.



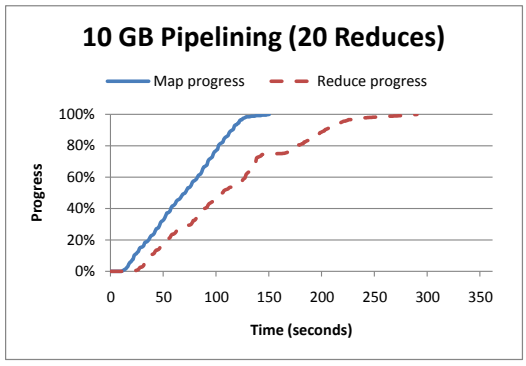


Figure 10.2: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 20 reduce tasks (512MB block size). The total job runtimes were 361 seconds for blocking and 290 seconds for pipelining.

"small" jobs used 10 worker nodes, while "large" jobs used 20. Each node was an "extra large" EC2 instances with 15GB of memory and four virtual cores, each running at 2.4GHz with a 2GB L2 cache.

Small Job Results

Our first experiment focused on the performance of small jobs in an underutilized cluster. We ran a 10GB wordcount with a 512MB block size, yielding 20 map tasks (one per block). We used 10 worker nodes and configured each worker to execute at most two map and two reduce tasks simultaneously. We ran several experiments to compare the performance of blocking and pipelining using different numbers of reduce tasks. For each experiment, we report the average progress over five separate runs.

Figure 10.1 reports the results of a job configured with five reduce tasks. A plateau can be seen at 75% progress for both blocking and pipelining. At this point in the job, all reduce tasks have completed the *shuffle* phase; the plateau is caused by the time taken to perform a final merge of all map output before entering the *reduce*

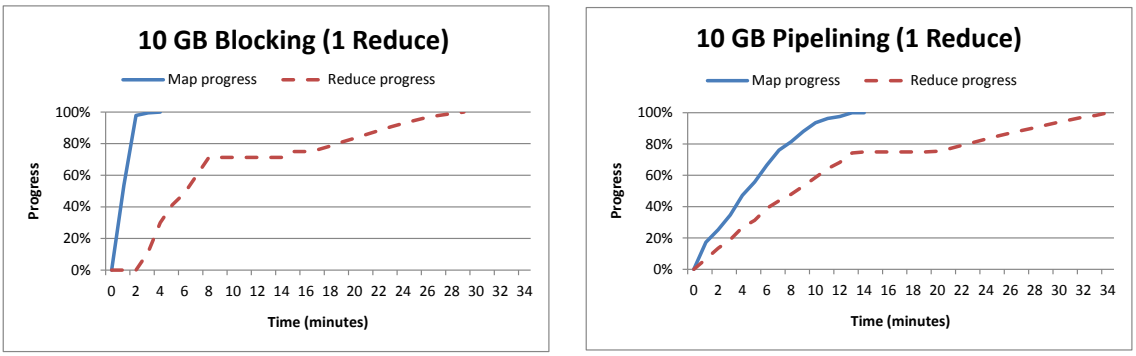


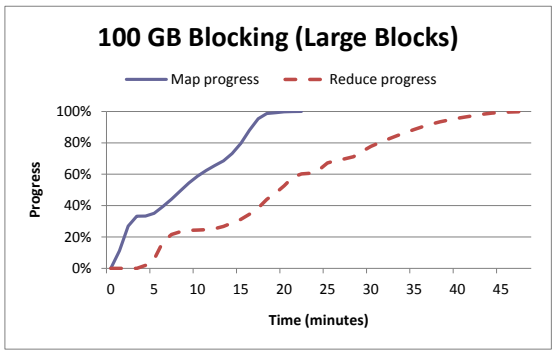
Figure 10.3: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 1 reduce task (512MB block size). The total job runtimes were 29 minutes for blocking and 34 minutes for pipelining.

phase. Notice that the plateau for the pipelining case is shorter. With pipelining, reduce tasks receive map outputs much earlier and can begin sorting earlier, thereby reducing the time required for the final merge.

Figure 10.2 reports the results with twenty reduce tasks. Using more reduce tasks decreases the amount of merging that any one reduce task must perform, which reduces the duration of the plateau at 75% progress. In the blocking case, the plateau is practically gone. However, with pipelining we still see a small plateau at 75% that, through further analysis using `iostat`, can be attributed to extra disk I/Os in the pipelining case. This extra memory pressure is due to diminished effectiveness of the combiner in the pipelining case. Although the response time of pipelining job is better than the blocking, a job that contains a more effective combiner may be better executed in blocking mode.

We further note that in both experiments, the map phase finishes faster with blocking than with pipelining. This is because pipelining allows reduce tasks to begin executing earlier and perform more work (sorting and combining); hence, the reduce tasks compete for resources with the map tasks, causing the map phase to take slightly longer. In this case, the increase in map duration is outweighed by the increase in cluster utilization, resulting in shorter job completion times: pipelining reduced completion time by 17.7% with 5 reducers and by 19.7% with 20 reducers.

Figure 10.3 describes an experiment in which we ran a 10GB wordcount job using a single reduce task. This caused job completion times to increase dramatically for both pipelining and blocking, because of the extreme load placed on the reduce node. Pipelining delayed job completion by about 17%, which suggests that our simple adaptive flow control scheme (Chapter 10.1.1) was unable to move load back to the map tasks aggressively enough in this (extremely) unbalanced job configuration.



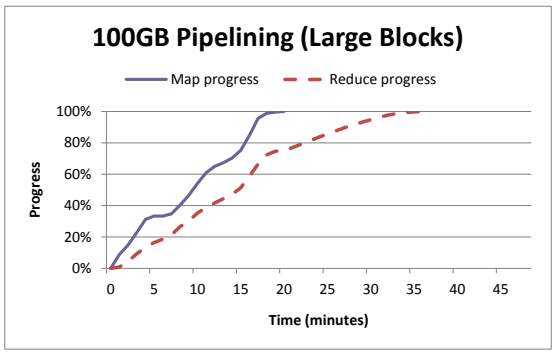
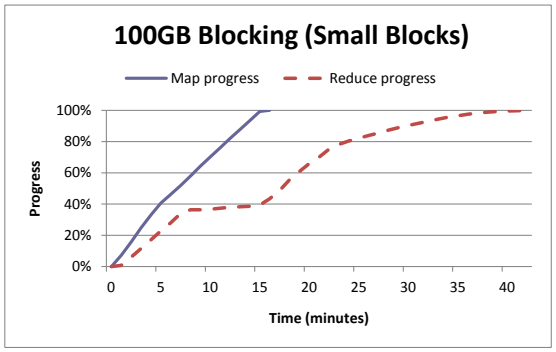


Figure 10.4: CDF of map and reduce task completion times for a 100GB wordcount job using 240 map tasks and 60 reduce tasks (512MB block size). The total job runtimes were 48 minutes for blocking and 36 minutes for pipelining.



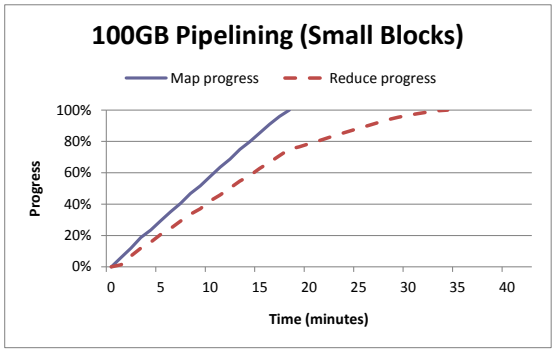


Figure 10.5: CDF of map and reduce task completion times for a 100GB wordcount job using 3120 map tasks and 60 reduce tasks (32MB block size). The total job runtimes were 42 minutes for blocking and 34 minutes for pipelining.

Large Job Results

Our second set of experiments focused on the performance of somewhat larger jobs. We increased the input size to 100GB (from 10GB) and the number of worker nodes to 20 (from 10). Each worker was configured to execute at most four map and three reduce tasks, which meant that at most 80 map and 60 reduce tasks could execute at once. We conducted two sets of experimental runs, each run comparing blocking to pipelining using either large (512MB) or small (32MB) block sizes. We were interested in blocking performance with small block sizes because blocking can effectively emulate pipelining if the block size is small enough.

Figure 10.4 reports the performance of a 100GB wordcount job with 512MB blocks, which resulted in 240 map tasks, scheduled in three waves of 80 tasks each. The 60 reduce tasks were co-scheduled with the first wave of map tasks. In the blocking case, the reduce tasks began working as soon as they received the output of the first wave, which is why the reduce progress begins to climb around four minutes (well before the completion of all maps). Pipelining was able to achieve significantly better cluster utilization, and hence reduced job completion time by about 25%.

Comparing the reduce progress in blocking to pipelining, we see that reduce tasks make more progress during the *shuffle* phase when pipelining is enabled. What is even more interesting is that the *reduce* phase is also shorter in the case of pipelining. The reason for this is subtle; all reduce tasks enter the *phase* around the same time since data is shipped in smaller increments. In the blocking case, when the final wave of map tasks finish they all try to send the entire output to reduce tasks at the same time, which increases the variance on receiving the complete output from all map tasks. That is, some reduce tasks enter the *reduce* phase well in advance of others.

Figure 10.5 reports the performance of blocking and pipelining using 32MB blocks. While the performance of pipelining remained similar, the performance of blocking improved considerably, but still trailed somewhat behind pipelining. Using block sizes smaller than 32MB did not yield a significant performance improvement in our experiments.

10.2 Online Aggregation

Although MapReduce was originally designed as a batch-oriented system, it is often used for interactive data analysis: a user submits a job to extract information from a data set, and then waits to view the results before proceeding with the next step in the data analysis process. This trend has accelerated with the development of high-level query languages that are executed as MapReduce jobs, such as Hive [91], Jaql [18], Pig [70], and Sawzall [74].

Traditional MapReduce implementations provide a poor interface for interactive data analysis, because they do not emit any output until the job has been executed to completion. In many cases, an interactive user would prefer a "quick and dirty" approximation over a correct answer that takes much longer to compute. In the database literature, online aggregation has been proposed to address this problem [45], but the batch-oriented nature of traditional MapReduce implementations makes these techniques difficult to apply. Here, we show how we extended our pipelined Hadoop implementation to support online aggregation within a single job (Chapter 10.2.1) and between multiple jobs (Chapter 10.2.2). In Chapter 10.2.3, we evaluate online aggregation on two different data sets, and show that it can yield an accurate approximate answer long before the job has finished executing.

10.2.1 Single-Job Online Aggregation

In HOP, the data records produced by map tasks are sent to reduce tasks shortly after each record is generated. However, to produce the final output of the job, the reduce function cannot be invoked until the entire output of every map task has been produced. We can support online aggregation by simply applying the reduce function to the data that a reduce task has received so far. We call the output of such an intermediate reduce operation a *snapshot*.

Users would like to know how accurate a snapshot is: that is, how closely a snapshot resembles the final output of the job. Accuracy estimation is a hard problem even for simple SQL queries [50], and particularly hard for jobs where the map and reduce functions are opaque user-defined code. Hence, we report job *progress*, not accuracy: we leave it to the user (or their MapReduce code) to correlate progress to a formal notion of accuracy. We define a simple progress metric later in this chapter.

Snapshots are computed periodically, as new data arrives at each reducer. The user specifies how often snapshots should be computed, using the progress metric as the unit of measure. For example, a user can request that a snapshot be computed when 25%, 50%, and 75% of the input has been seen. The user may also specify whether to include data from tentative (unfinished) map tasks. This option does not affect the fault-tolerance design described in Chapter 10.1.3. In the current prototype, each snapshot is stored in a directory on HDFS. The name of the directory includes the progress value associated with the snapshot. Each reduce task runs independently, and at a different rate. Once a reduce task has made sufficient progress, it writes a snapshot to a temporary directory on HDFS, and then atomically renames it to the appropriate location.

Applications can consume snapshots by polling HDFS in a predictable location. An application knows that a given snapshot has been completed when every reduce task has written a file to the snapshot directory. Atomic rename is used to avoid applications mistakenly reading incomplete snapshot files.

Note that if there are not enough free slots to allow all the reduce tasks in a job to be scheduled, snapshots will not be available for reduce tasks that are still waiting to be executed. The user can detect this situation (e.g., by checking for the expected number of files in the HDFS snapshot directory), so there is no risk of incorrect data, but the usefulness of online aggregation will be reduced. In the current prototype, we manually configured the cluster to avoid this scenario. The system could also be enhanced to avoid this pitfall entirely by optionally waiting to execute an online aggregation job until there are enough reduce slots available.

Progress Metric

Hadoop provides support for monitoring the progress of task executions. As each map task executes, it is assigned a *progress score* in the range [0,1], based on how much of its input the map task has consumed. We reused this feature to determine how much progress is represented by the current input to a reduce task, and hence to decide when a new snapshot should be taken. When the transfer of a spill file to a reduce task occurs, we include a small amount of meta-data that indicates the map's current progress score, relative to that spill file. To compute the overall progress score for a reduce step snapshot, we take the average of the progress scores associated with each map's data residing on the reduce task prior to executing the snapshot.

Note that it is possible to have a map task that has not pipelined *any* output to a

reduce task, either because the map task has not been scheduled yet (there are no free TaskTracker slots), the map tasks does not produce any output for the given reduce task, or because the reduce task has been configured to only pipeline data from at most k map tasks concurrently. To account for this, we need to scale the progress metric to reflect the portion of the map tasks that a reduce task has pipelined data from: if a reducer is connected to $\frac{1}{n}$ of the total number of map tasks in the job, we divide the average progress score by n.

This progress metric could easily be made more sophisticated: for example, an improved metric might include the selectivity ($|output|/|input|$) of each map task, the statistical distribution of the map task's output, and the effectiveness of each map task's combine function, if any. Although we have found our simple progress metric to be sufficient for most experiments we describe below, this clearly represents an opportunity for future work.

10.2.2 Multi-Job Online Aggregation

Online aggregation is particularly useful when applied to a long-running analysis task composed of multiple MapReduce jobs. As described in Chapter 10.1.2, our version of Hadoop allows the output of a reduce task to be sent directly to map tasks. This feature can be used to support online aggregation for a sequence of jobs.

Suppose that j_1 and j_2 are two MapReduce jobs, and j_2 consumes the output of j_1. When j_1's reducers compute a snapshot to perform online aggregation, that snapshot is written to HDFS, and also sent directly to the map tasks of j_2. The map and reduce steps for j_2 are then computed as normal, to produce a snapshot of j_2's output. This process can then be continued to support online aggregation for an arbitrarily long sequence of jobs.

Unfortunately, inter-job online aggregation has some drawbacks. First, the output of a reduce function is not "monotonic": the output of a reduce function on the first 50% of the input data may not be obviously related to the output of the reduce function on the first 25%. Thus, as new snapshots are produced by j_1, j_2 must be recomputed from scratch using the new snapshot. As with inter-job pipelining (Chapter 10.1.2), this could be optimized for reduce functions that are declared to be distributive or algebraic aggregates [39].

To support fault-tolerance for multi-job online aggregation, we consider three cases. Tasks that fail in j_1 recover as described in Chapter 10.1.3. If a task in j_2 fails, the system simply restarts the failed task. Since subsequent snapshots produced by j_1 are taken from a superset of the mapper output in j_1, the next snapshot received by the restarted reduce task in j_2 will have a higher progress score. To handle failures in j_1, tasks in j_2 cache the most recent snapshot received by j_1, and replace it when they receive a new snapshot with a higher progress metric. If tasks from both jobs fail, a new task in j_2 recovers the most recent snapshot from j_1 that was stored in HDFS and then wait for snapshots with a higher progress score.

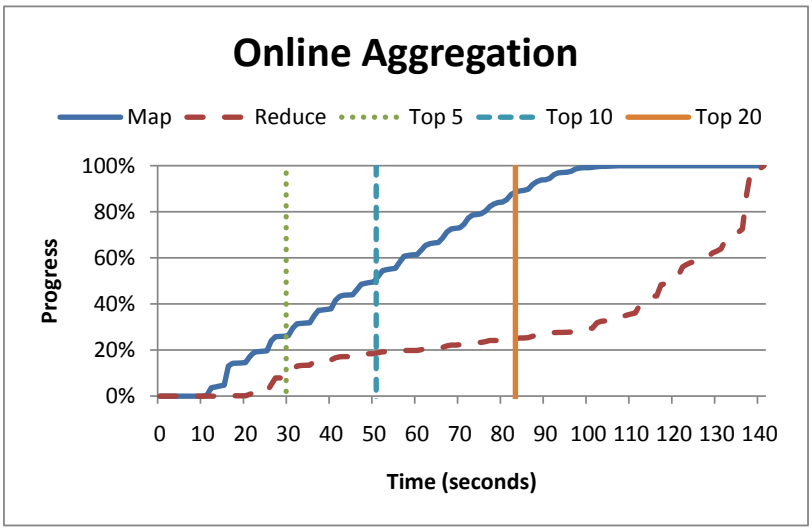


Figure 10.6: Top-100 query over 5.5GB of Wikipedia article text. The vertical lines describe the increasing accuracy of the approximate answers produced by online aggregation.

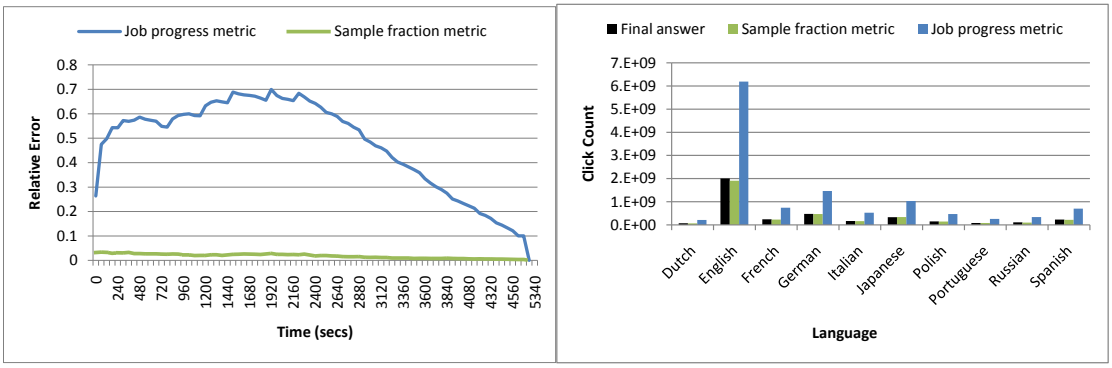
10.2.3 Evaluation

To evaluate the effectiveness of online aggregation, we performed two experiments on Amazon EC2 using different data sets and query workloads. In our first experiment, we wrote a "Top-K" query using two MapReduce jobs: the first job counts the frequency of each word and the second job selects the K most frequent words. We ran this workload on 5.5GB of Wikipedia article text stored in HDFS, using a 128MB block size. We used a 60-node EC2 cluster; each node was a "high-CPU medium" EC2 instance with 1.7GB of RAM and 2 virtual cores. A virtual core is the equivalent of a 2007-era 2.5Ghz Intel Xeon processor. A single EC2 node executed the Hadoop JobTracker and the HDFS NameNode, while the remaining nodes served as slaves for running the TaskTrackers and HDFS DataNodes.

Figure 10.6 shows the results of inter-job online aggregation for a Top-100 query. Our accuracy metric for this experiment is post-hoc — we note the time at which the Top-K words in the snapshot are the Top-K words in the final result. Although the final result for this job did not appear until nearly the end, we did observe the Top-5, 10, and 20 values at the times indicated in the graph. The Wikipedia data set was biased toward these Top-K words (e.g., "the", "is", etc.), which remained in their correct position throughout the lifetime of the job.

Approximation Metrics

In our second experiment, we considered the effectiveness of the job progress metric described in Chapter 10.2.1. Unsurprisingly, this metric can be inaccurate when it is used to estimate the accuracy of the approximate answers produced by online aggregation. In this experiment, we compared the job progress metric with a simple



(a) Relative approximation error over time

(b) Example approximate answer

Figure 10.7: Comparison of two approximation metrics. Figure (a) shows the relative error for each approximation metric over the runtime of the job, averaged over all groups. Figure (b) compares an example approximate answer produced by each metric with the final answer, for each language and for a single hour.

user-defined metric that leverages knowledge of the query and data set. HOP allows such metrics, although developing such a custom metric imposes more burden on the programmer than using the generic progress-based metric.

We used a data set containing seven months of hourly page view statistics for more than 2.5 million Wikipedia articles [88]. This constituted 320GB of compressed data (1TB uncompressed), divided into 5066 compressed files. We stored the data set on HDFS and assigned a single map task to each file, which was decompressed before the map function was applied.

We wrote a MapReduce job to count the total number of page views for each language and each hour of the day. In other words, our query grouped by language and hour of day, and summed the number of page views that occurred in each group. To enable more accurate approximate answers, we modified the map function to include the fraction of a given hour that each record represents. The reduce function summed these fractions for a given hour, which equated to one for all records from a single map task. Since the total number of hours was known ahead of time, we could use the result of this sum over all map outputs to determine the total fraction of each hour that had been sampled. We call this user-defined metric the "sample fraction."

To compute approximate answers, each intermediate result was scaled up using two different metrics: the generic metric based on job progress and the sample fraction described above. Figure 10.7a reports the relative error of the two metrics, averaged over all groups. Figure 10.7b shows an example approximate answer for a single hour using both metrics (computed two minutes into the job runtime). This figure also contains the final answer for comparison. Both results indicate that the sample fraction metric provides a much more accurate approximate answer for this query than the progress-based metric.

Job progress is clearly the wrong metric to use for approximating the final an-

swer of this query. The primary reason is that it is too coarse of a metric. Each intermediate result was computed from some fraction of each hour. However, the job progress assumes that this fraction is uniform across all hours, when in fact we could have received much more of one hour and much less of another. This assumption of uniformity in the job progress resulted in a significant approximation error. By contrast, the sample fraction scales the approximate answer for each group according to the actual fraction of data seen for that group, yielding much more accurate approximations.

10.3 Continuous Queries

MapReduce is often used to analyze streams of constantly-arriving data, such as URL access logs [28] and system console logs [99]. Because of traditional constraints on MapReduce, this is done in large batches that can only provide periodic views of activity. This introduces significant latency into a data analysis process that ideally should run in near-real time. It is also potentially inefficient: each new MapReduce job does not have access to the computational state of the last analysis run, so this state must be recomputed from scratch. The programmer can manually save the state of each job and then reload it for the next analysis operation, but this is labor-intensive.

Our pipelined version of Hadoop allows an alternative architecture: MapReduce jobs that run *continuously*, accepting new data as it becomes available and analyzing it immediately. This allows for near-real-time analysis of data streams, and thus allows the MapReduce programming model to be applied to domains such as environment monitoring and real-time fraud detection.

In this section, we describe how HOP supports continuous MapReduce jobs, and how we used this feature to implement a rudimentary cluster monitoring tool.

10.3.1 Continuous MapReduce Jobs

A bare-bones implementation of continuous MapReduce jobs is easy to implement using pipelining. No changes are needed to implement continuous map tasks: map output is already delivered to the appropriate reduce task shortly after it is generated. We added an optional "flush" API that allows map functions to force their current output to reduce tasks. When a reduce task is unable to accept such data, the mapper framework stores it locally and sends it at a later time. With proper scheduling of reducers, this API allows a map task to ensure that an output record is promptly sent to the appropriate reducer.

To support continuous reduce tasks, the user-defined reduce function must be periodically invoked on the map output available at that reducer. Applications will have different requirements for how frequently the reduce function should be invoked; pos-

sible choices include periods based on wall-clock time, logical time (e.g., the value of a field in the map task output), and the number of input rows delivered to the reducer. The output of the reduce function can be written to HDFS, as in our implementation of online aggregation. However, other choices are possible; our prototype system monitoring application (described below) sends an alert via email if an anomalous situation is detected.

In our current implementation, the number of map and reduce tasks is fixed, and must be configured by the user. This is clearly problematic: manual configuration is error-prone, and many stream processing applications exhibit "bursty" traffic patterns, in which peak load far exceeds average load. In the future, we plan to add support for elastic scaleup/scaledown of map and reduce tasks in response to variations in load.

Fault Tolerance

In the checkpoint/restart fault-tolerance model used by Hadoop, mappers retain their output until the end of the job to facilitate fast recovery from reducer failures. In a continuous query context, this is infeasible, since mapper history is in principle unbounded. However, many continuous reduce functions (e.g., 30-second moving average) only require a suffix of the map output stream. This common case can be supported easily, by extending the JobTracker interface to capture a rolling notion of reducer consumption. Map-side spill files are maintained in a ring buffer with unique IDs for spill files over time. When a reducer commits an output to HDFS, it informs the JobTracker about the *run* of map output records it no longer needs, identifying the run by spill file IDs and offsets within those files. The JobTracker can then tell mappers to garbage collect the appropriate data.

In principle, complex reducers may depend on very long (or infinite) histories of map records to accurately reconstruct their internal state. In that case, deleting spill files from the map-side ring buffer will result in potentially inaccurate recovery after faults. Such scenarios can be handled by having reducers checkpoint internal state to HDFS, along with markers for the mapper offsets at which the internal state was checkpointed. The MapReduce framework can be extended with APIs to help with state serialization and offset management, but it still presents a programming burden on the user to correctly identify the sensitive internal state. That burden can be avoided by more heavyweight process-pair techniques for fault-tolerance, but those are quite complex and use significant resources [83]. In our work to date we have focused on cases where reducers can be recovered from a reasonable-sized history at the mappers, favoring minor extensions to the simple fault-tolerance approach used in Hadoop.

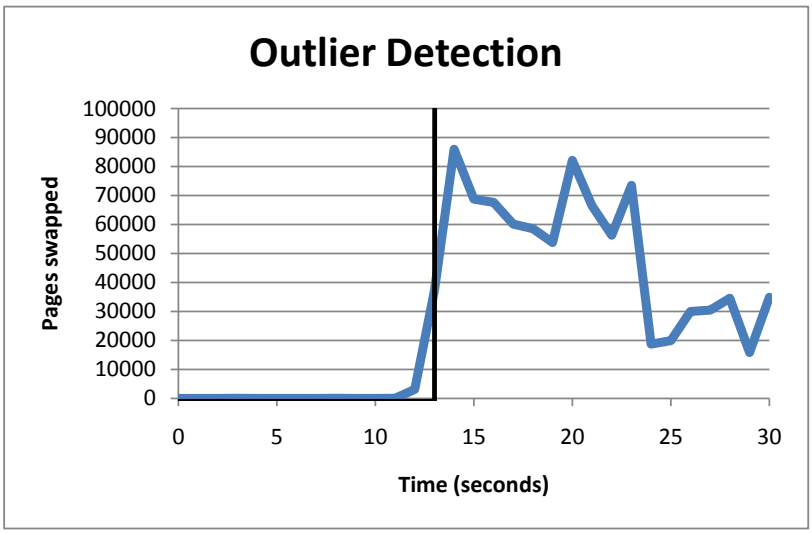


Figure 10.8: Number of pages swapped over time on the thrashing host, as reported by `vmstat`. The vertical line indicates the time at which the alert was sent by the monitoring system.

10.3.2 Prototype Monitoring System

Our monitoring system is composed of *agents* that run on each monitored machine and record statistics of interest (e.g., load average, I/O operations per second, etc.). Each agent is implemented as a continuous map task: rather than reading from HDFS, the map task instead reads from various system-local data streams (e.g., `/proc`).

Each agent forwards statistics to an *aggregator* that is implemented as a continuous reduce task. The aggregator records how agent-local statistics evolve over time (e.g., by computing windowed-averages), and compares statistics between agents to detect anomalous behavior. Each aggregator monitors the agents that report to it, but might also report statistical summaries to another "upstream" aggregator. For example, the system might be configured to have an aggregator for each rack and then a second level of aggregators that compare statistics between racks to analyze datacenter-wide behavior.

Evaluation

To validate our prototype system monitoring tool, we constructed a scenario in which one member of a MapReduce cluster begins thrashing during the execution of a job. Our goal was to test how quickly our monitoring system would detect this behavior. The basic mechanism is similar to an alert system one of the authors implemented at an Internet search company.

We used a simple load metric (a linear combination of CPU utilization, paging, and swap activity). The continuous reduce function maintains windows over samples of this metric: at regular intervals, it compares the 20 second moving average of the

load metric for each host to the 120 second moving average of all the hosts in the cluster *except* that host. If the given host's load metric is more than two standard deviations above the global average, it is considered an outlier and a tentative alert is issued. To dampen false positives in "bursty" load scenarios, we do not issue an alert until we have received 10 tentative alerts within a time window.

We deployed this system on an EC2 cluster consisting of 7 "large" nodes (large nodes were chosen because EC2 allocates an entire physical host machine to them). We ran a wordcount job on the 5.5GB Wikipedia data set, using 5 map tasks and 2 reduce tasks (1 task per host). After the job had been running for about 10 seconds, we selected a node running a task and launched a program that induced thrashing.

We report detection latency in Figure 10.8. The vertical bar indicates the time at which the monitoring tool fired a (non-tentative) alert. The thrashing host was detected very rapidly—notably faster than the 5-second TaskTracker-JobTracker heartbeat cycle that is used to detect straggler tasks in stock Hadoop. We envision using these alerts to do early detection of stragglers within a MapReduce job: HOP could make scheduling decisions for a job by running a secondary continuous monitoring query. Compared to out-of-band monitoring tools, this economy of mechanism—reusing the MapReduce infrastructure for reflective monitoring—has benefits in software maintenance and system management.

10.4 BOOM-MR Port

This chapter describes our port of the BOOM-MR (Chapter 9) to HOP. Using BOOM-MR, we developed alternative scheduling policies, written in Overlog, that made use of statistics provided by the monitoring system described in Chapter 10.3.2. In Chapter 10.4.1, we describe the port of JOL to the HOP JobTracker scheduling component. Chapter 10.5 describes the interface between the monitoring system and JOL, which enables the use of the monitoring results in our declarative scheduling logic. In Chapter 10.5.4, we present an Overlog rule that monitors tasks for anomalous behavior [26]; spawning a backup/speculative task when alerted to a potential issue.

10.4.1 Scheduling HOP with JOL

HOP is based on Hadoop 19.2, which defines an extensible interface to the JobTracker scheduler component for alternative scheduler implementations. This made the port of JOL to HOP trivial: the entire port consisted of 55 lines of Java glue code that implemented the JOL harness, and Overlog code that performed the basic FIFO policy described in Chapter 9. We altered the job relation (described in Table 9.1) to include an attribute for the job type: pipelining/blocking, online aggregation, or continuous. We also added three new scheduling rules (presented in Figure 10.10) specific to online aggregation and continuous jobs.

```
public abstract class TaskScheduler implements Configurable {
   ...

  public abstract List<Task>
   assignTasks(TaskTrackerStatus taskTracker) throws IOException;
}
```

Figure 10.9: Task scheduler interface. Not all methods shown.

JOL Port

In Hadoop 19.2, the JobTracker makes use of an interface called the *TaskScheduler* to implement alternative task scheduling policies. Figure 10.9 shows a partial view of this interface, which contains the method `assignTasks` that is passed a TaskTracker status object and returns a list of tasks that should be scheduled. This method is called by the JobTracker during a *heartbeat* exchange with a TaskTracker.

Our implementation of the `assignTasks` method transforms the TaskTracker status object into a tuple that updates the `taskTracker` relation in Table 9.1. In response to this update, the scheduling rules enter a fixpoint computation, during which it may assign task attempts to the given TaskTracker. Any updates to the *schedule* relation (see rule `s5` in Figure 9.2) will trigger a (pre-registered) Java listener that translates the update into a *Task* object, which the `assignTasks` method accumulates in a *List* object that is returned by the `assignTasks` method at the end of the fixpoint.

Job submission interface

The Hadoop JobTracker interface for submitting jobs had to be retrofitted to support pipelining between jobs. In regular Hadoop, jobs are submitted one at a time; a job that consumes the output of one or more other jobs cannot be submitted until the producer jobs have completed. To support this, we modified the Hadoop job submission interface to accept a list of jobs, where each job in the list depends on the job before it. The client interface traverses this list, annotating each job with the identifier of the job that it depends on. We then added a new table to the declarative scheduler that captured inter-job dependencies. The job scheduling rules use this table to co-schedule jobs with their dependencies, giving slot preference to "upstream" jobs over the "downstream" jobs they feed. As we note in Chapter 11, there are many interesting options for scheduling pipelines or even DAGs of such jobs that we plan to investigate in future.

Online aggregation and continuous job scheduling policies

Online aggregation and continuous jobs rely on a scheduling policy that ensures the execution of the entire pipeline. In the case of online aggregation, a more complete pipeline provides more accurate estimates since unscheduled partitions (i.e., groups)

```
h1 unscheduledReduceTasks(JobId, a_count<TaskId>) :-
   job(JobId, JType, ...),
   task(JobId, TaskId, TType, Status, ...),
   JType == JobType.ONLINE, TType == TaskType.REDUCE,
   Status.state() != TaskState.RUNNING;

h2 canScheduleMaps(JobId) :-
   unscheduledReduceTasks(JobId, Count),
   Count == 0;

h3 canScheduleMaps(JobId) :-
   job(JobId, Type, ...),
   Type != JobType.ONLINE;
```

Figure 10.10: Counts the number of reduce tasks that are not running and only schedules map tasks from an online job when this count is zero.

may contain important data. For continuous jobs, scheduling the entire pipeline is a requirement in order to avoid the memory pressure in storing the (continuously arriving) data for an unscheduled operator. We enforced this constraint with by a policy that scheduled reduce tasks before any map tasks in the same job (assuming sufficient slot capacity).

Figure 10.10 shows three rules that together determine when a job is allowed to schedule map tasks. A separate admission controller rule ensured that the number of reduce tasks for an online aggregation or continuous job fit within the current cluster-wide slot capacity. For each job, rule `h1` counts the number of reduce tasks not currently running. If the job type is "online" then rule `h2` will add the fact that map tasks can be scheduled when the number of non-running map tasks is equal to zero. Rule `h3` applies to the map tasks in all other job types; it simply removes this scheduling constraints on those map tasks. The `canScheduleMaps` predicate is included in the rule that determines the scheduling of map tasks (e.g., rule `s4` in Figure 9.2).

10.5 Real-time monitoring with JOL

After porting BOOM-MR to HOP, we started writing scheduler policies based on the real-time monitoring information supplied by our monitoring job. In order to do this, we needed to import the results of our MapReduce monitoring job into JOL as relations. Here, we further describe the MapReduce job that continuously monitors HOP and its interface to JOL. We then present an alert system that detects outlier measurements in map and reduce task execution attempts. We conclude our discussion with a new task speculation policy that is based on our alert system.

Measure	*Description*	*Source*
COMP_EST	Task estimated completion time	Overlog
USER_CPU	User CPU usage	/proc/stat, /proc/[pid]/stat
SYS_CPU	System CPU usage	/proc/stat, /proc/[pid]/stat
RSS	Resident set memory size	/proc/[pid]/stat
VSIZE	Estimated completion time	/proc/[pid]/stat
WRITE_BYTES	Number of bytes written	/proc/[pid]/io
READ_BYTES	Number of bytes read	/proc/[pid]/io
NET_OUT	Network output	/proc/net/dev
NET_IN	Network input	/proc/net/dev
SWAP_OUT	Swaps out	/proc/vmstat
SWAP_IN	Swaps in	/proc/vmstat
PAGE_OUT	Pages out	/proc/vmstat
PAGE_IN	Pages in	/proc/vmstat

Table 10.1: HOP monitoring measurements.

10.5.1 MapReduce monitoring job

The MapReduce job that monitors HOP is scheduled during the system bootstrap. The job executes a single map task on each TaskTracker in the cluster and some number (based on the size of the cluster) of reduce tasks that group machine and rack level statistics. For example, we could schedule a single reduce task per rack that aggregates the statistics gathered on that rack.

Table 10.1 lists the measurements that we collected. The measurement name is given in the first column, followed by a measurement description. The last column identifies the location under `/proc` where the measurement was taken. Process level measurements reside under `/proc/[pid]/`, where `[pid]` represents the process identifier. All other measurements outside of `/proc/[pid]/` refer to machine level measurements with the exception of the estimated completion time, which is derived from task level statistics in the JobTracker.

A map task gathers measurements by periodically reading the source location (last column in Table 10.1) from the local file system. For each measurement, the map task outputs a record `<host name, time stamp, pid, measurement name, measurement value>`. For machine statistics, the map task will set the PID field to $\underline{0}$ e.g., `<boom.cs.berkeley.edu, 12348234, 0, NET_OUT, 101>`. The record key for all map outputs is the identifier of the rack to which the machine belongs. If the cluster does not contain rack-level information then the host name is used instead. This ensures that a single reduce task will see all measurements from a given rack or machine boundary.

Name	*Description*	*Relevant attributes*
machineStat	Machine statistics	Host, Measure, TimeStamp, Value
proccessStat	Process statistics	TaskId, Pid, Measure, TimeStamp, Value
jobStat	Job statistics	JobId, TaskType, Measure, StatContainer
taskStat	Task statistics	JobId, TaskId, Measure, TaskType, Value
alert	Outlier task alerts	TaskId, TimeStamp, Measure, Description, Severity

Table 10.2: JOL monitoring relations.

```
/* Correlate process measurements to the actual map/reduce task */
ts1 taskStat(TaskId.getJobID(), TaskId, Measure, Type, TimeStamp,
             Value) :-
    taskAttempt(TaskId, ... , TaskState.RUNNING, Pid),
    processStat(Host, Pid, Measure, TimeStamp, Value),
    Type := TaskId.isMap() ? TaskType.MAP : TaskType.REDUCE;

/* Compute the estimated completion time based on the task rate
   of progress */
ts2 taskStat(JobId, TaskId, COMP_EST, TaskType, TimeStamp, CompEst) :-
    taskAttempt(TaskId, ... , Progress, ProgressRate, TaskState.RUNNING,
                Pid),
    JobId := TaskId.getJobID(),
    Type := TaskId.isMap() ? TaskType.MAP : TaskType.REDUCE,
    CompEst := ProgressRate == 0 ? infinity :
               (1f - Progress) / ProgressRate,
    TimeStamp := java.lang.System.currentTimeMillis();
```

Figure 10.11: Rules for maintaining the `taskStat` table.

10.5.2 Monitoring with Overlog

The output of the monitoring job is sent directly — reduce tasks open a back-channel TCP-socket — to the JOL instance running on the JobTracker. The receiver code translates the data packets into JOL tuples, and inserts them into monitoring relations defined in Table 10.2. The `machineStat` and `processStat` tables are populated by the data packets received from the monitoring jobs. The `jobStat` and `taskStat` tables maintain statistics for jobs and tasks, respectively, and are derived by Overlog rules (Figures 10.11 and 10.12). The `alert` table contains outlier task measurements, which depending on the severity can result in corrective action e.g., execute a speculative task (Chapter 10.5.4).

Figure 10.11 contains two rules that together maintain the `taskStat` table. The `taskAttempt` table was extended to include the task process identifier (*Pid*), which is supplied by the TaskTracker executing the task attempt. The process identifier allows us to correlate a task in the `taskAttempt` table with process level measurements in the `processStat` table, as shown by rule `ts1`. A task's estimated completion time is

```
js1 taskStatList(JobId, TaskType, Measure, a_list<Value>) :-
    taskStat(JobId, TaskId, Measure, TaskType, TimeStamp, Value);


js2 jobStat(JobId, TaskType, Measure, Statistics) :-
    taskStatList(JobId, TaskType, Measure, TaskStatList),
    Statistics := new StatContainer(TaskStatList);
```

Figure 10.12: Rules for maintaining the `jobStat` relation.

```
a1 alert (TaskId, TimeStamp, Measure, Desc, Severity) :-
   taskStat(JobId, TaskId, Measure, TaskType, TimeStamp, TaskStat),
   jobStat(JobId, TaskType, Measure, JobStat),
   JobStat.outlier(Measure, TaskStat) == true,
   Desc := JobStat.description(Measure, TaskStat),
   Severity := JobStat.severity(Measure, TaskStat),
   TimeStamp := java.lang.System.currentTimeMillis();
```

Figure 10.13: Rule for detecting outlier tasks.

based on its current progress and progress rate: change in progress computed over TaskTracker heartbeat intervals. Using the current progress and progress rate, rule `ts2` computes a rough estimate on the remaining time it will take for the task to complete, which we have denoted as a COMP_EST measurement — stored in the `taskStat` table.

Figure 10.12 contains the rules for maintaining the `jobStat` table. The `taskStatList` table, maintained by rule `js1`, provides a list of measurement values for each job identifier, task type, and measurement name. The `jobStat` table groups measurement values belonging to the same job, task type, and measurement name. A special Java object called *StatContainer* is used to store each group of measurements. The *StatContainer* class defines methods for computing various metrics (e.g., mean, median, stddev, etc.) from its list of measurement values. Rule `js2` maintains the `jobStat` table by initializing a *StatContainer* object for each group of aggregated measurement values.

10.5.3 Task alerts

Figure 10.13 contains a single rule that detects outlier task by correlating the task measurement with information in the `jobStat` table. We compare the measurements from tasks that belong to the same category — job and task type (map or reduce). The *JobStat* variable references a *StatContainer* object for a given category, and it is used to determine if a task belonging to that category is an outlier based on some metric e.g., k deviations from the mean. The *JobStat* variable is also used to provide a description and severity of outlier measurement.

```
s1 mostRecentCriticalAlert(TaskId, Measure, a_min<AlertTime>) :-
   alert(TaskId, AlertTime, Measure, Desc, Severity),
   Severity.contains(``critical'');  /* The alert is critical */

s2 schedule(Tracker, list<TaskId, MapSlots>) :-
   heartbeat(Tracker, TrackerStatus, MapSlots, _),
   MapSlots > 0,
   mostRecentCriticalAlert(TaskId, Measure, AlertTime)

   /* Ensure the alert is not too old (alert time < 10 seconds ago). */
   (java.lang.System.currentTimeMillis() - AlertTime) < 10000,

   /* The task's estimated time to completion is very high relative
      to equivalent tasks. */
   taskStat(JobId, TaskId, COMP_EST, TaskType, TimeStamp, TaskStat),
   jobStat(JobId, TaskType, COMP_EST, JobStat),
   TaskStat < JobStat.percentile(0.25),

   /* Schedule backup map task if host has split AND
      no backup task has yet been scheduled */
   task(JobId, TaskId, ..., Splits, ...), TaskId.isMap(),
   taskAttemptCount(TaskId, Count), Count == 1,
   InputSplits.contains(TrackerStatus.getHost());
```

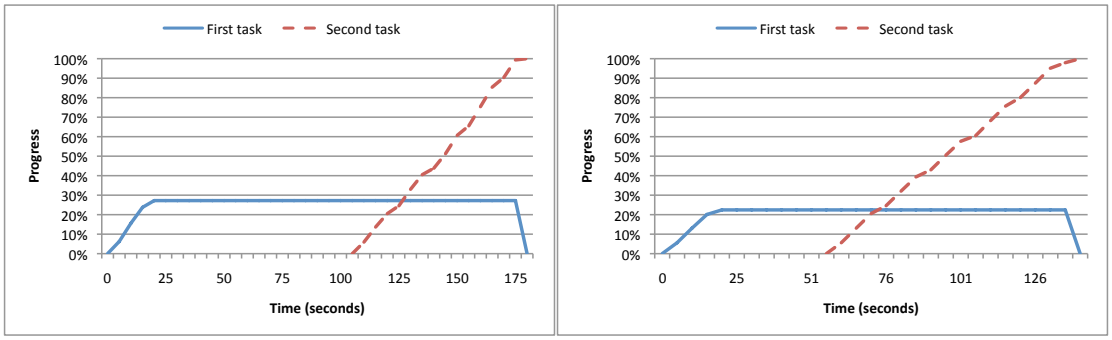
Figure 10.14: Rule for map task speculation based on alert system data. Reduce task speculation rule is similar (we do no consider splits) and therefore omitted.

10.5.4 Alert based speculation policy

Figure 10.14 contains a rule that reschedules map tasks with any "critical" alerts that occurred recently; rule `s1` defines the `mostRecentCriticalAlert` relation. Rule `s2` is evaluated at the JobTracker whenever a *heartbeat* exchange occurs with some TaskTracker. The `heartbeat` predicate includes the name of the TaskTracker, its status, and its spare map slot capacity, which the rule ensures is greater than zero. The rule joins the `heartbeat` with all critical alerts in the `mostRecentCriticalAlert` relation. As an added precaution, we subsequently check that the alerted task's estimated completion (`COMP_EST`) time is high relative to other tasks in its category. Finally, we ensure that the task has not already been rescheduled and that the TaskTracker contains the maps input data.

10.5.5 Evaluation

We compared our alert based speculation policy with the speculation policy implemented in unmodified Hadoop 19.2. Our experiment executed a wordcount job that contained a single faulty map task that would execute normally for a minute before stalling out by sleeping for one second intervals between map function invocations.



(a) Hadoop 19.2 task speculation policy (b) HOP alert based task speculation policy

Figure 10.15: Compares speculation policies by plotting the starting point and progress of the faulty task (first task) and speculative task (second task).

The input to the wordcount was 10GB of randomly generated words, yielding 20 map tasks total. We executed this job on a 20 node EC2 cluster and compared the time it took to initiate a speculative task using our policy to the policy in unmodified Hadoop.

Figure 10.15 shows the result of this experiment by plotting the launch time and progress of the original (first) task and the backup (second) task. HOP's alert based speculation policy is able to detect the faulty map task and execute a backup task in half the time of unmodified Hadoop. In unmodified Hadoop, a task is speculated based on its rate of progress (relative to other tasks in its category). We are able to further extend this policy by including machine and process level statistics as further evidence to speculate. Indeed, our choice to speculate was not only based on a high estimated time to completion but also a critically low "user CPU" value and a critically low I/O activity.

The astute reader will notice however that the rate of progress for the second task in HOP is less than that of unmodified Hadoop. The reason for this is that our monitoring jobs do add some extra load to the cluster. Nevertheless, in this instance, the overall job response time was slightly less (a few seconds) in HOP due to the faster turn around time in our speculation policy.

10.6 Related Work

This work relates to literature on parallel dataflow frameworks, online aggregation, and continuous query processing.

10.6.1 Parallel Dataflow

Dean and Ghemawat's paper on Google's MapReduce [28] has become a standard reference, and forms the basis of the open-source Hadoop implementation. The Google MapReduce design targets very large clusters where the probability of worker failure or slowdown is high. This led to their elegant checkpoint/restart approach to fault-tolerance, and their lack of pipelining. Our work extends the Google design to accommodate pipelining without significant modification to their core programming model or fault tolerance mechanisms.

Dryad [47] is a data-parallel programming model and runtime that is often compared to MapReduce, supporting a more general model of acyclic dataflow graphs. Like MapReduce, Dryad puts disk materialization steps between dataflow stages by default, breaking pipelines. The Dryad paper describes support for optionally "encapsulating" multiple asynchronous stages into a single process so they can pipeline, but this requires a more complicated programming interface.

It has been noted that parallel database systems have long provided partitioned dataflow frameworks [72], and recent commercial databases have begun to offer MapReduce programming models on top of those frameworks [87, 40]. Most parallel database systems can provide pipelined execution akin to our work here, but they use a more tightly coupled iterator and *Exchange* model that keeps producers and consumers rate-matched via queues, spreading the work of each dataflow stage across all nodes in the cluster [35]. This provides less scheduling flexibility than MapReduce and typically offers no tolerance to mid-query worker faults. Yang et al. recently proposed a scheme to add support for mid-query fault-tolerance to traditional parallel databases, using a middleware-based approach that shares some similarities with MapReduce [100].

Logothetis and Yocum describe a MapReduce interface over a continuous query system called *Mortar* that is similar in some ways to our work [58]. Like HOP, their mappers push data to reducers in a pipelined fashion. They focus on specific issues in efficient stream query processing, including minimization of work for aggregates in overlapping windows via special reducer APIs. They are not built on Hadoop, and explicitly sidestep issues in fault-tolerance.

Hadoop Streaming is part of the Hadoop distribution, and allows map and reduce functions to be expressed as UNIX shell command lines. It does not stream data through map and reduce phases in a pipelined fashion.

10.6.2 Online Aggregation

Online aggregation was originally proposed in the context of simple single-table SQL queries involving "Group By" aggregations, a workload quite similar to MapReduce [45]. The focus of the initial work was on providing not only "early returns" to these SQL queries, but also statistically robust estimators and confidence interval

metrics for the final result based on random sampling. These statistical matters do not generalize to arbitrary MapReduce jobs, though our framework can support those that have been developed. Subsequently, online aggregation was extended to handle join queries (via the *Ripple Join* method), and the *CONTROL* project generalized the idea of online query processing to provide interactivity for data cleaning, data mining, and data visualization tasks [44]. That work was targeted at single-processor systems. Luo et al. developed a partitioned-parallel variant of Ripple Join, without statistical guarantees on approximate answers [65].

In recent years, this topic has seen renewed interest, starting with Jermaine et al.'s work on the *DBO* system [50]. That effort includes more disk-conscious online join algorithms, as well as techniques for maintaining randomly-shuffled files to remove any potential for statistical bias in scans [49]. Wu et al. describe a system for peer-to-peer online aggregation in a distributed hash table context [98]. The open programmability and fault-tolerance of MapReduce are not addressed significantly in prior work on online aggregation.

An alternative to online aggregation combines precomputation with sampling, storing fixed samples and summaries to provide small storage footprints and interactive performance [34]. An advantage of these techniques is that they are compatible with both pipelining and blocking models of MapReduce. The downside of these techniques is that they do not allow users to choose the query stopping points or time/accuracy trade-offs dynamically [44].

10.6.3 Continuous Queries

In the last decade there was a great deal of work in the database research community on the topic of continuous queries over data streams, including systems such as Borealis [4], STREAM [67], and Telegraph [22]. Of these, Borealis and Telegraph [83] studied fault-tolerance and load balancing across machines. In the Borealis context this was done for pipelined dataflows, but without partitioned parallelism: each stage ("operator") of the pipeline runs serially on a different machine in the wide area, and fault-tolerance deals with failures of entire operators [14]. SBON [73] is an overlay network that can be integrated with Borealis, which handles "operator placement" optimizations for these wide-area pipelined dataflows.

Telegraph's *FLuX* operator [83, 84] is the only work to our knowledge that addresses mid-stream fault-tolerance for dataflows that are both pipelined and partitioned in the style of HOP. FLuX ("Fault-tolerant, Load-balanced eXchange") is a dataflow operator that encapsulates the shuffling done between stages such as map and reduce. It provides load-balancing interfaces that can migrate operator state (e.g., reducer state) between nodes, while handling scheduling policy and changes to data-routing policies [84]. For fault-tolerance, FLuX develops a solution based on process pairs [83], which work redundantly to ensure that operator state is always being maintained live on multiple nodes. This removes any burden on the continuous query programmer of the sort we describe in Chapter 10.3. On the other hand, the FLuX

protocol is far more complex and resource-intensive than our pipelined adaptation of Google's checkpoint/restart tolerance model.

10.7 Summary

In this chapter, we extended the batch-oriented execution model of MapReduce to support pipelining between operators. This enables a new suite of MapReduce jobs that are able to perform online aggregation and continuous like queries. Unlike much of the work on online aggregation, we do not focus here on statistical guarantees because of the flexibility of the MapReduce programming model. These guarantees are crafted for specific SQL aggregates like SUMs, COUNTs, and AVERAGEs, and modified to account for processing techniques like the join algorithms used. The focus of our work here is architectural: to provide "early returns" interactions within the powerful scalability and fault-tolerance facilities of MapReduce frameworks. The statistical guarantees from the literature only apply to SQL-style reduce functions; statistical guarantees for other online reducers would need to be developed in a case-by-case basis. We expect that in many cases users will settle for simply observing changes in the output of a job over time, and make their own decisions about whether early returns are sufficient.

We leveraged our ability to run continuous MapReduce jobs in HOP by developing a monitoring framework that provides near real-time machine and process level statistics. Our monitoring framework enabled new scheduling opportunities that are based on such statistics. Porting the declarative scheduler to HOP allowed us to quickly prototype alternative policies in Overlog where, in many cases, adding new scheduling constraints translated into adding/removing a few rule predicates.

Chapter 11

Conclusion and Future Extensions

Declarative programming allows programmers to focus on the high level properties of a computation without describing low level implementation details. We have found that declarative programming not only simplifies a programmer's work it also focuses the programming task on the appropriate high-level issues. The declarative networking project exemplified this through its declarative specifications of network protocols that could execute on either wired or wireless physical networks. It was the responsibility of the compiler to take these simple high-level specifications and map them to an underlining technology.

The Evita Raced meta-compilation framework takes declarative programming a step further by allowing Overlog program transformations to be written in Overlog and executed by the P2 query processing engine. The use of metacompilation allowed us to achieve significant code reuse from the core of P2, so that the mechanisms supporting query optimization are a small addition to the query processing functionality already in the system. A particularly elegant aspect of this is the scheduling of independent optimization stages by expressing scheduling constraints as data, and having that data processed by a special dataflow element for scheduling. Our hypothesis that a Datalog-style language was a good fit for typical query optimizations was largely borne out, despite some immaturity in the Overlog language and P2 infrastructure. We were able to express three of the most important optimizer frameworks — System R, Cascades, and Magic-sets — in only a few dozen rules each.

Our experience developing BOOM Analytics in Overlog resulted in a number of observations that are useful on both long and short timescales. Some of these may be specific to our BOOM agenda of rethinking programming frameworks for distributed systems; a number of them are more portable lessons about distributed system design that apply across programming frameworks.

At a high level, the effort convinced us that a declarative language like Overlog is practical and beneficial for implementing substantial systems infrastructure, not just the isolated protocols tackled in prior work. Though our metrics were necessarily rough (code size, programmer-hours), we were convinced by the order-of-magnitude

improvements in programmer productivity, and more importantly by our ability to quickly extend our implementation with substantial new distributed features. Performance remains one of our concerns, but not an overriding one. One simple lesson of our experience is that modern hardware enables "real systems" to be implemented in very high-level languages. We should use that luxury to implement systems in a manner that is simpler to design, debug, secure and extend — especially for tricky and mission-critical software like distributed services.

We have tried to separate the benefits of data-centric system design from our use of a high-level declarative language. Our experience suggests that data-centric programming can be useful even when combined with a traditional programming language, particularly if that language supports set-oriented data processing primitives (e.g., LINQ, list comprehensions). Since traditional languages do not necessarily encourage data-centric programming, the development of libraries and tools to support this design style is a promising direction for future work.

Moving forward, our experience highlighted problems with Overlog that emphasize some new research challenges; we mention two here briefly. First, and most urgent, is the need to codify the semantics of asynchronous computations and updateable state in a declarative language. Recent follow on work has made some progress on defining a semantic foundation for this [9], and initial efforts at a programmer-friendly language [8]. A second key challenge is to clarify the implementation of invariants, both local and global. In an ideal declarative language, the specification of an invariant should entail its automatic implementation. In our experience with Overlog this was hampered both by the need to explicitly write protocols to test global invariants, and the multitude of possible mechanisms for enforcing invariants, be they local or global. A better understanding of the design space for invariant detection and enforcement would be of substantial use in building distributed systems, which are often defined by such invariants.

MapReduce is another example of raising the level of abstraction to the programming task of coordinating a computation on a large number of machine. Our Hadoop Online Prototype extends the applicability of the model to pipelining behaviors, while preserving the simple programming model and fault tolerance of a full-featured MapReduce framework. This provides significant new functionality, including "early returns" on long-running jobs via online aggregation, and continuous queries over streaming data. We also demonstrate benefits for batch processing: by pipelining both within and across jobs, HOP can reduce the time to job completion.

In considering future work, scheduling is a topic that arises immediately. Stock Hadoop already has many degrees of freedom in scheduling batch tasks across machines and time, and the introduction of pipelining in HOP only increases this design space. First, pipeline parallelism is a new option for improving performance of MapReduce jobs, but needs to be integrated intelligently with both intra-task partition parallelism and speculative redundant execution for "straggler" handling. Second, the ability to schedule deep pipelines with direct communication between reduces and maps (bypassing the distributed file system) opens up new opportunities

and challenges in carefully co-locating tasks from different jobs, to avoid communication when possible.

Olston and colleagues have noted that MapReduce systems — unlike traditional databases — employ "model-light" optimization approaches that gather and react to performance information during runtime [69]. The continuous query facilities of HOP enable powerful introspective programming interfaces for this: a full-featured MapReduce interface can be used to script performance monitoring tasks that gather system-wide information in near-real-time, enabling tight feedback loops for scheduling and dataflow optimization. This is a topic we plan to explore further, including opportunistic methods to do monitoring work with minimal interference to outstanding jobs, as well as dynamic approaches to continuous optimization in the spirit of earlier work like Eddies [13] and FLuX [84].

Online aggregation changes some of the scheduling criteria in cases where there are not enough slots systemwide for all of a job's tasks. Map and reduce tasks affect an online aggregation job differently: leaving map tasks unscheduled is akin to sampling the input file, whereas leaving reduce tasks unscheduled is akin to missing certain output keys – some of which could be from groups with many inputs. This favors reducers over mappers, at least during early stages of processing.

In order to improve early results of pipelined flows (e.g., for online aggregation), it is often desirable to prioritize "interesting" data in the pipeline, both at the mapper and reducer. Online reordering of data streams has been studied in the centralized setting [78], but it is unclear how to expose it in the MapReduce programming framework, with multiple nodes running in parallel – especially if the data in the input file is not well randomized.

Continuous queries over streams raise many specific opportunities for optimizations, including sharing of work across queries on the same streams, and minimizing the work done per query depending on windowing and aggregate function semantics. Many of these issues were previously considered for tightly controlled declarative languages on single machines [67, 22], or for wide-area pipelined dataflows [4, 73], and would need to be rethought in the context of a programmable MapReduce framework for clusters.

As a more long-term agenda, we want to explore using MapReduce-style programming for even more interactive applications. As a first step, we hope to revisit interactive data processing in the spirit of the CONTROL work [44], with an eye toward improved scalability via parallelism. More aggressively, we are considering the idea of bridging the gap between MapReduce dataflow programming and lightweight event-flow programming models like SEDA [95]. Our HOP implementation's roots in Hadoop make it unlikely to compete with something like SEDA in terms of raw performance. However, it would be interesting to translate ideas across these two traditionally separate programming models, perhaps with an eye toward building a new and more general-purpose declarative framework for programming in architectures like cloud computing and many-core.

Bibliography

[1] Bison—GNU parser generator. `http://www.gnu.org/software/bison/`. Fetched on 11/15/2007.

[2] Flex—The Fast Lexical Analyzer. `http://www.gnu.org/software/flex/manual/`. Fetched on 11/15/2007.

[3] Java programming lanuage.

[4] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Conference on Innovative Data Systems Research (CIDR)*, January 2005.

[5] Martín Abadi and Boon Thau Loo. Towards a Declarative Language and System for Secure Networking. In *International Workshop on Networking Meets Databases (NetDB)*, 2007.

[6] Azza Abouzeid et al. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB*, 2009.

[7] Peter Alvaro, Tyson Condie, Neil Conway, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. BOOM: Data-centric programming in the datacenter. In *EuroSys*, 2010.

[8] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[9] Peter Alvaro et al. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

[10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[11] Michael P. Ashley-Rollman, Michael De Rosa, Siddhartha S. Srinivasa, Padmanabhan Pillai, Seth Copen Goldstein, and Jason D. Campbell. Declarative Programming for Modular Robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications*, 2007.

[12] Michael P. Ashley-Rollman et al. Declarative Programming for Modular Robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications*, 2007.

[13] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2000.

[14] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD*, 2005.

[15] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM.

[16] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *J. Log. Program.*, 10:255–299, March 1991.

[17] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[18] K. Beyer, V. Ercegovac, and E. Shekita. Jaql: A json query language.

[19] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[20] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[21] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[22] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[23] Yanpei Chen, Laura Keys, and Randy H. Katz. Towards energy efficient mapreduce. Technical Report UCB/EECS-2009-109, EECS Department, University of California, Berkeley, Aug 2009.

[24] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The Design and Implementation of a Declarative Sensor Network System. In *SenSys*, 2007.

[25] Yang-Hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *Proc. of ACM SIGMETRICS*, pages 1–12, 2000.

[26] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online aggregation and continuous query support in mapreduce. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1115–1118, New York, NY, USA, 2010. ACM.

[27] Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, and Sean Rheaand Timothy Roscoe. Finally, a use for componentized transport protocols. In *HotNets IV*, 2005.

[28] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[29] Giuseppe DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[30] Jason Eisner et al. Dyna: a declarative language for implementing dynamic programs. In *ACL*, 2004.

[31] Apache Software Foundation, Isabel Drost, Ted Dunning, Jeff Eastman, Otis Gospodnetic, Grant Ingersoll, Jake Mannix, Sean Owen, and Karl Wettin. Apache mahout, 2010. `http://mloss.org/software/view/144/`.

[32] D.J. DeWitt G. Graefe. The EXODUS Optimizer Generator. In *SIGMOD*, 1987.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[34] Phillip B. Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD*, 1998.

[35] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, 1990.

[36] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3), 1995.

[37] Goetz Graefe. Iterators, schedulers, and distributed-memory parallelism. *Softw. Pract. Exper.*, 26(4), 1996.

[38] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, 1993.

[39] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[40] Greenplum. A unified engine for RDBMS and MapReduce, 2009. `http://www.greenplum.com/resources/mapreduce/`.

[41] Haryadi S. Gunawi et al. SQCK: A Declarative File System Checker. In *OSDI*, 2008.

[42] Hadoop jira issue tracker, July 2009. `http://issues.apache.org/jira/browse/HADOOP`.

[43] Hadoop jira issue tracker, July 2009. `http://issues.apache.org/jira/browse/HADOOP`.

[44] Joseph M. Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J. Haas. Interactive data analysis with CONTROL. *IEEE Computer*, 32(8), August 1999.

[45] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD*, 1997.

[46] Ryan Huebsch, Brent Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The architecture of PIER: an internet-scale query processor. In *Conference on Innovative Data Systems Research (CIDR)*, 2005.

[47] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[48] Michael Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[49] Chris Jermaine. Online random shuffling of large database tables. *IEEE Trans. Knowl. Data Eng.*, 19(1):73–84, 2007.

[50] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable approximate query processing with the DBO engine. In *SIGMOD*, 2007.

[51] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[52] Navin Kabra and David J. Dewitt. Opt++: An object-oriented implementation for extensible database query optimization. *VLDB Journal*, 8:55–78, 1999.

[53] Eddie Kohler et al. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[54] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[55] Monica S. Lam et al. Context-sensitive program analysis as database queries. In *PODS*, 2005.

[56] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-Sensitive Program Analysis as Database Queries. In *PODS*, 2005.

[57] N. Li and J.C. Mitchell. Datalog with Constraints: A Foundation for Trust-management Languages. In *International Symposium on Practical Aspects of Declarative Languages*, 2003.

[58] Dionysios Logothetis and Kenneth Yocum. Ad-hoc data processing in the cloud (demonstration). *Proc. VLDB Endow.*, 1(2), 2008.

[59] Guy Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *SIGMOD*, 1988.

[60] S Lohr. Google and ibm join in cloud computing research, 2007.

[61] Boon Thau Loo. *The Design and Implementation of Declarative Networks*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2006.

[62] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*, 2006.

[63] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Proc. ACM SOSP*, October 2005.

[64] Yu-En Lu. *Distributed Proximity Query Processing*. PhD thesis, University of Cambridge, Cambridge, UK, 2007. Under review.

[65] Gang Luo, Curt J. Ellmann, Peter J. Haas, and Jeffrey F. Naughton. A scalable hash ripple join algorithm. In *SIGMOD*, 2002.

[66] Katherine A. Morris, Jeffrey D. Ullman, and Allen Van Gelder. Design overview of the nail! system. In *ICLP*, pages 554–568, 1986.

[67] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.

[68] Nokia Corporation. disco: massive data – minimal code, 2009. `http://discoproject.org/`.

[69] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Technical Conference*, 2008.

[70] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[71] Owen O'Malley. Hadoop map/reduce architecture, July 2006. Presentation, `http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/HadoopMapReduceArch.pdf`.

[72] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. Dewitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[73] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proc.International Conference on Data Engineering (ICDE)*, 2006.

[74] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[75] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule-Based Query Rewrite Optimization in Starburst. In *SIGMOD*, 1992.

[76] R. Ramakrishnan and J. Ullman. A survey of research in deductive database systems. Technical Report 1995-14, Stanford Infolab, 1995.

[77] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[78] Vijayshankar Raman, Bhaskaran Raman, and Joseph M. Hellerstein. Online dynamic reordering for interactive data processing. In *In VLDB*, 1999.

[79] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[80] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems, 1975.

[81] Thorsten Schutt et al. Scalaris: Reliable transactional P2P key/value store. In *ACM SIGPLAN Workshop on Erlang*, 2008.

[82] Patricia G. Selinger, M. Astrahan, D. Chamberlin, Raymond Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 1979.

[83] Mehul A. Shah, Joseph M. Hellerstein, and Eric A. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *SIGMOD*, 2004.

[84] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.

[85] Ehud Y. Shapiro. Systems programming in concurrent prolog. In *POPL*, pages 93–105, 1984.

[86] Leonard Shapiro, Yubo Fan, Yu Zhang, David Maier, Paul Benninghoff, Kavita Hatwal, Hsiao min Wu, Keith Billings, Quan Wang, and Bennet Vance. Exploiting upper and lower bounds in top-down query optimization. In *In Proceedings of IDEAS 01*, 2001.

[87] Ajeet Singh. Aster *n*Cluster in-database MapReduce: Deriving deep insights from large datasets, 2009. `http://www.asterdata.com/resources/downloads/whitepapers/Aster_MapReduce_Technical_Whitepaper.pdf`.

[88] Peter N. Skomoroch. Wikipedia page traffic statistics, 2009. Downloaded from `http://developer.amazonwebservices.com/connect/entry.jspa?externalID=2596`.

[89] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[90] Michael Stonebraker and Greg Kemnitz. The POSTGRES next generation database management system. *Commun. ACM*, 34(10):78–92, 1991.

[91] Ashish Thusoo et al. Hive - a warehousing solution over a Map-Reduce framework. In *VLDB*, 2009.

[92] Shalom Tsur and Carlo Zaniolo. Ldl: A logic-based data language. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 33–41, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[93] Jeffrey D. Ullman. Lecture Notes on the Magic-Sets Algorithm. `http://infolab.stanford.edu/~ullman/cs345notes/slides01-16.pdf`. Fetched on 11/15/2007.

[94] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Company, 1990.

[95] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

[96] Walker White, Alan Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling games to epic proportions. In *Proc. SIGMOD*, 2007.

[97] Walker White et al. Scaling games to epic proportions. In *SIGMOD*, 2007.

[98] Sai Wu, Shouxu Jiang, Beng Chin Ooi, and Kian-Lee Tan. Distributed online aggregation. In *VLDB*, 2009.

[99] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.

[100] Christopher Yang, Christine Yen, Ceryen Tan, and Samuel Madden. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *ICDE*, 2010.

[101] Fan Yang et al. Hilda: A high-level language for data-driven web applications. In *ICDE*, 2006.

[102] Yuan Yu et al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[103] Matei Zaharia et al. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[104] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Y Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *In Proc. of the 8th Symposium on Operating Systems Design and Implementation (OSDI 08) (San Diego CA)*, 2008.