Evita Raced: Metacompilation for Declarative Networks

*Tyson Condie, *David Chu, Joseph M. Hellerstein, and Petros Maniatis
UC Berkeley and Intel Research Berkeley

ABSTRACT

Declarative languages have recently been proposed for many new applications outside of traditional data management. Since these are relatively early research efforts, it is important that the architectures of these declarative systems be extensible, in order to accommodate unforeseen needs in these new domains. In this paper, we apply the lessons of declarative systems to the *internals* of a declarative engine. Specifically, we describe our design and implementation of *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. We describe the minimalist architecture of Evita Raced, including its extensibility interfaces and its reuse of P2's data model and runtime engine. We demonstrate that a declarative language like OverLog is well-suited to expressing traditional and novel query optimizations as well as other query manipulations, in a compact and natural fashion. Finally, we present initial results of Evita Raced extended with various optimization programs, running on both Internet overlay networks and wireless sensor networks.

1. INTRODUCTION

There has been renewed interest in recent years in applying declarative programming to a variety of applications outside the traditional boundaries of data management. Examples include work on compilers [20], computer games [35], natural language processing [9], security protocols [21], information extraction [31] and modular robotics [3]. Our own work in this area has focused on *Declarative Networking*, as instantiated in the *P2* system for Internet overlays [24, 25], and the *DSN* system for wireless sensor networks [6]; this work has been extended by various colleagues as well [1, 5, 32, 33].

There is a strong analogy between the Internet today, and database systems in the 1960's. Network protocol implementations involve complex procedural code, and there is increasing need to separate their specification from physical and logical changes to components underneath them: network fabrics and architectures are in a period of swift evolution [2]. Hence the lessons of Data Independence and declarative approaches are very timely in this domain [15], and are reflected by recent interest in automatic network optimization and adaptation [11]. Moreover, we have observed that many networking tasks are naturally described in recursive query languages like Datalog, because (a) they typically involve recursive graph traversals (e.g., shortest-path computations) [26], and (b) the asynchronous messaging inherent in networks is neatly expressed as joins of message streams with "rendezvous" or "session" tables [24, 25].

Given these intuitions, we implemented the P2 and DSN systems, and demonstrated the utility of the declarative approach with Datalog-based implementations of a host of network functionalities at various levels of the protocol stack. Both of these systems allow protocols to be expressed as programs in a Datalog-like language, which are compiled to dataflow runtime implementations reminiscent of traditional database query plans. We have found that using a declarative language often results in drastic reductions in code size (100× and more) relative to procedural languages like C++. Perhaps more surprising, our declarative protocols are often quite intuitive: in many cases they are almost line-for-line translations of published pseudocode, suggesting that Datalog is indeed a good match for the application domain [6, 25].

1.1 A Reflection on Declarative Languages

Declarative Networking and related topics have the potential to expand the impact of database research into new domains, while reviving interest in classical database topics like recursive query processing that had received minimal attention in recent years. Yet our own systems are implemented in imperative programming languages: the P2 declarative overlay system is implemented in C++, and the DSN declarative sensor network system is implemented in an embedded dialect of C. We recently began asking ourselves whether Codd's vision applies to our own efforts: can declarative programming improve the implementation of declarative systems?

In this paper, we put declarative systems "in the mirror," investigating a declarative implementation of a key aspect of a declarative system. Specifically, we have reimplemented the query planning component of P2 as a *metacompiler*: a compiler for the P2 language, OverLog, that is itself written in OverLog. We call the resulting implementation "Evita Raced"[1]. We use Evita Raced primarily for query optimization, extending P2 with a number of important query optimization techniques it formerly lacked. Our experience has been quite positive: we were able to relatively easily take P2 from having almost no optimizations at all, to having a

[*] Tyson Condie and David Chu are supported in part by the National Science Foundation under Grants IIS-0713661 and CNS-0722077, and by a gift from Microsoft Corporation.

VLDB '08, August 24-30, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

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

fairly sophisticated (and growing) set of optimizations. For example, we implemented the traditional System R dynamic programming optimizer (including choice of access and join methods, "interesting orders," and histograms) in only 51 OverLog rules (292 lines of code); our implementation of the Supplementary Magic Sets rewriting optimization for recursive queries [34] is not only compact (68 rules, 264 lines), but also a close translation of the description from Ullman's textbook on the subject [34].

The elegance of our approach comes in part from the fact that query optimization techniques—like many search algorithms—are at heart recursive algorithms, and benefit from a declarative recursive language in much the same way as routing protocols. Even non-recursive optimization logic—such as parts of Ullman's magic-sets pseudocode—is simple enough to express in a declarative fashion that abstracts away mechanistic details such as the scheduling of data-parallel steps (e.g., scanning all rules in a program in parallel versus sequentially).

Our contributions here are three-fold. First, we present a declarative architecture for query optimization that is based on metacompilation, reusing the query executor in a stylized fashion to serve as the engine beneath the optimization process. This results in an *economy of mechanism* [28] not afforded by earlier extensible optimizers. Second, we show that a variety of traditional and novel query optimizations are easy to express in a recursive, declarative language. Finally, we validate the simplicity and applicability of our design via an implementation of an OverLog query optimizer for the P2 Declarative Networking engine, which also cross-compiles to DSN programs that run on wireless sensor networks. Based on our experience to date, 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 remain unclear.

2. P2: LANGUAGE AND ARCHITECTURE

We begin our discussion with an overview of key aspects of the P2 declarative network system. 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. P2 supports a recursive query language called OverLog that resembles traditional Datalog with some extensions we discuss below. Each P2 node runs the same runtime engine, and, by default, participates equally in every declarative program. In parallel programming terms, P2 supports the Single-Program-Multiple-Data (SPMD) model of parallel computation.

The P2 engine at each node consists of a compiler—which parses programs and physically plans them—a dataflow runtime engine, and access methods. The original P2 compiler was monolithic and implemented in an imperative language (C++). The subject of this work is the replacement of that monolithic compiler with a runtime-extensible compiler framework that admits declaratively-specified optimizations, as well as compilation stages that perform functions other than performance optimization. In this section, we highlight the distinguishing features of the OverLog language, as well as the P2 dataflow executor and access methods.

2.1 OverLog, Revisited

Our original paper on P2 presented OverLog in an ad-hoc manner as an event-driven language [25]. In a subsequent paper, we provided a syntax and declarative semantics for a subset of OverLog called *Network Datalog* that supported distributed Datalog programs with aggregation that could be laid out on network links [24]. Since that time, we have modified the OverLog language and the P2 runtime semantics a fair bit in an attempt to merge practical matters from networking with declarative semantics. Much of this work is documented in Loo's dissertation [23], the remainder is reflected in the public P2 code release. In this section we overview the current semantics of OverLog that are relevant to our discussion here.

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

link(@"localhost:10000", "localhost:10001").
link(@"localhost:10001", "localhost:10002").
...

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

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

r4 minCostPath(@X,Y,a_min<C>) :-
          path(@X,Y,P,C).

r5 shortestPath(@X,Y,P,C) :-
          minCostPath(@X,Y,C), path(@X,Y,P,C).

Query: shortestPath(@LOCALHOST,"localhost:10000",P,C).
```

Figure 1: Shortest path program in OverLog. `a_` **prefixes introduce aggregate functions and** `f_` **prefixes introduce built-in functions.**

OverLog is based on the traditional recursive query language, Datalog; we assume a passing familiarity with Datalog in our discussion. As in Datalog, an OverLog *program* consists of a collection of deduction *rules* that define the set of all derived tuples from a base set of tuples called *facts* (see Figure 1 for an example OverLog program used in what follows). 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*[2]. OverLog introduces some notable extensions to Datalog that we describe next, to illustrate in more detail the particular language for which we are writing a metacompiler.

Horizontal partitioning—As in the Network Datalog description mentioned above [24], OverLog's basic data model consists of relational tables that are partitioned across the nodes in a P2 network. Each relation in an OverLog rule must have one attribute that 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 describe 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. Syntactic tests and transformations ("localization") ensure that a set of rules can be maintained in a manner consistent with its location specifiers and network topology [24], in environments that lack all-to-all network connectivity. We reimplemented localization declaratively in

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

Evita Raced, via 28 OverLog rules (104 lines).

Soft State, Events, and Fixpoints—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 infinity. Zero-lifetime tables are referred to as *event* tables, and their tuples are called *events*; all other tables are referred to as *materialized* tables. OverLog contains a `materialize` declaration that specifies the lifetime of a materialized table. In OverLog, events are defined to occur one-at-a-time *between* fixpoint computations at a given node. Hence each fixpoint computation on the OverLog rules operates with a traditional, static set of stored tuples: (a) the local tuples in materialized tables whose lifetime has not run out, (b) at most one local event fact across *all* event tables (the *current* event being processed), and (c) any derived local tuples that can be deduced from (a) and (b) via the program rules. This captures the semantics of OverLog on each network node individually. As of the time of writing, P2 only provides semantic guarantees across nodes in the network for monotonic OverLog programs (without negation or aggregation) [24], and for "local-only" programs that never repartition data. Extensions on this front are an open area of research, but not directly relevant to Evita Raced, which currently consists of local-only rules.

Stratification—Like many Datalog systems, P2 only supports programs whose use of negation and aggregation is stratified [34], that is, there is no aggregation or negation on a recursive cycle of head/body rule dependencies. Checking this property is very easy in Evita Raced; it is a transitive closure program on the rule graph, which we implemented in 5 OverLog rules (27 lines).

Deletions and Updates—Like SQL, OverLog supports set-oriented expressions that identify tuples to be deleted or updated. To this end, any OverLog rule in a program can be prefaced by the keyword `delete`. The `delete` rule body specifies facts to be deleted. In addition to deletes, OverLog's `materialize` statement supports the specification of a primary key for each relation, and these relations can appear in the heads of rules. Any derived tuple for that relation that matches an existing tuple on the primary key is intended to *replace* that existing tuple. For semantic simplicity, OverLog deletions are defined to occur only after the fixpoint computation of the program that generates them.

Status—In the interest of full disclosure, we note that the current P2 release has a design flaw in the deferral of processing certain tuples, which compromises its ability to achieve two aspects of the described semantics. Specifically, it may (a) process multiple strata on one node at the same time, and (b) remove `delete` tuples from materialized tables before the end of the fixpoint that generates them. This flaw does not directly affect the work described in this paper. The fix for this problem has been designed, but was not implemented at the time of publication.

2.1.1 A Canonical Example

To illustrate the specifics of OverLog, we briefly revisit a shortest-paths example (Figure 1). The three `materialize` statements specify that `link`, `path` and `shortestPath` are all tables with infinite lifetime and infinite storage space[3]. 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 any 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 consequently 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 `Z`, the output of which is to be stored back at `X`. Such rules can be easily rewritten (localized) so that the body predicates all have the same location specifier [24, 26]; the only communication then is shipping the results of the deduction to the head relation's location specifier.

2.2 The P2 Runtime Engine

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 [18]. Like Click, the P2 runtime supports dataflow *elements* (or "operators") of two sorts: pull-based elements akin to database iterators [13], and 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 [25].

2.2.1 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. 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). To this suite, we added sorting and merge-joins, which allow us to explore some traditional query optimization opportunities and trade offs as discussed in Section 4.1.

Given its focus on network protocols and soft state, P2 currently has no support for persistent storage other than 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 query predicates.

P2 also 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 [7]. The P2 planner currently assembles these network elements in a fixed manner so 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.

2.2.2 The P2 Event Loop

The control flow in the P2 runtime is driven by a fairly traditional event loop that responds to any network or timer event by invoking an appropriate dataflow segment to handle the event.

The basic control loop in P2 works as follows:

1. An event is taken from the system input queue, corresponding to a single newly arrived tuple to be inserted in a table. We will refer to this tuple as the *current event* tuple.
2. The value of the system clock is noted in a variable we will call the *current time*. Soft-state tuples whose lifetime is over as of the current time are skipped (and removed from internal storage) during subsequent processing.
3. The current event tuple is, logically, appended to its table.
4. The dataflow corresponding to the OverLog program is initiated and runs to a local fixpoint following traditional Datalog

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

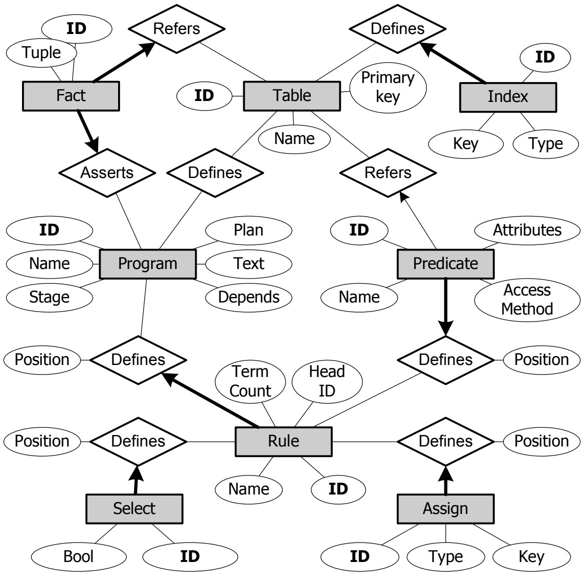

Figure 2: ER Diagram of a query plan in P2.

semantics, with the following exception: during processing, any non-local derived tuples are buffered in a *send queue*; local deletions are postponed until the end of the fixpoint.

5. Upon fixpoint completion, the contents of the send queue are transmitted over the network, and at that point any buffered local tuple deletions are performed.

As noted above, our current prototype lacks deferred deletions.

3. DECLARATIVE COMPILATION

Evita Raced is a compiler (i.e., query optimizer and rewriter) for OverLog that supports a runtime-extensible set of program rewrites and optimizations, which can themselves be expressed in OverLog. A key contribution of this work is the economy of mechanism that we achieve via the use of P2's dataflow runtime system to implement the compiler. Two main challenges must be addressed to make this work. First, all compiler state—the internal representation of declarative OverLog programs—needs to 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. Both challenges must be addressed without hindering the function of those rewrites and optimizations that are implemented in more traditional, imperative code (e.g., in C++).

In this section 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 fragments needed to bootstrap the optimizer.

3.1 Table-izing Optimizer State

A typical query optimizer maintains a number of data structures to describe the contents of a query, and to represent ongoing properties of a query planner including fragments of query plans. Our first task in designing Evita Raced was to capture this information in a relational schema.

Figure 2 shows an Entity-Relationship diagram that captures the properties of an OverLog program. We derived the constraints in the diagram directly from 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 1) 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 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 a number of *facts* at startup, each of which must belong to a single program and must reference a single table.

Table 1: The Metacompiler Catalog: tables defining an OverLog program and dataflow execution plan.

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	Predicates appearing in a rule	**id**, **rule_id**, table_id, name, position, access_method
select	Selections appearing in a rule	**id**, **rule_id**, boolean, position
assign	Assignment statements appearing in a rule	**id**, **rule_id**, variable, value, position

The conversion of the ER diagram to relational format is straightforward. Table 1 lists the relations that capture the entities 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 optimization programs that get added to the system. The primary key columns are bold in Figure 2 and Table 1.

3.2 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. In this section we describe how Evita Raced represents query optimization programs as dataflow, and also the way it orchestrates multiple different optimization programs through the P2 dataflow framework.

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. One way to write an Evita Raced stage is to construct a monolithic program in C++, and encapsulate it in a single dataflow element; this is how we implement certain base stages required for bootstrapping, as described in Section 3.3. However, the power of Evita Raced comes from its support for stages written in OverLog, which, in addition to being compactly expressed in a high-level language, can be loaded into a running P2 installation at any time. A stage programmer registers a new stage with Evita Raced by inserting a tuple into the *program* relation. This tuple contains a unique identifier ($program\_id$), a name ($name$), a list of stage dependencies ($depends$) , and the program text ($text$). Because the `program` relation is used to convey partial compilation results from stage to stage as well, `program` tuples also contain attributes for the name of the compiler stage operating on the program ($stage$), and the stage's final physical plan ($plan$), though these attributes are empty when the programmer first creates the tuple. Section 3.2.2 describes the $depends$ attribute, and its use in the installation of new compiler stages. The $plan$ attribute pertains to the physical planner stage, which is described in Section 3.3.2. The *program* table is also used to store users' OverLog programs (not compiler stages); for these programs, the $depends$ attribute must be empty. We next describe the interfaces to an Evita Raced compiler stage, after which we discuss the way that multiple such stages are coordinated.

3.2.1 The Stage API

At base, 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 stage runs its dataflow over that event and the tables in the Metacompiler Catalog, typically modifying catalog tables in some way, until it inserts a new `program` tuple, containing the name of the stage in the $stage$ attribute, into the program table. This insertion indicates the completion of the stage.

To represent this behavior in a OverLog stage, a relatively simple template can be followed. An OverLog stage must have at least one rule body containing the `<stage>::programEvent` predicate. This represents the ability of the stage to react to new programs arriving at the system. In addition, the stage must have at least one rule with a `program` head predicate, which derives a new `program` tuple when signaling stage completion. OverLog stages may be recursive programs, so they run to fixpoint before completing.

3.2.2 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 Catalogs, and an *Installer* stage must follow all other stages, since by installing the dataflow program into the P2 runtime it terminates a compilation session. We will see other specific precedence constraints in Section 4.

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 arbitrary partial 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 (we review built-in stages in Section 3.3).

To achieve the dataflow connections among stages, the built-in *StageScheduler* element listens for updates to the `program` table, indicating the arrival of a new OverLog program or the completion of a stage for an on-going program compilation, as described in the previous section. The *StageScheduler* is responsible for shepherding stage execution according to the `StageLattice`. Given a `program` update, it joins it with the lattice to identify next stages that can be invoked, and generates a `<stage>::programEvent` tuple that will start that stage; the contents of these tuples are the same as those of the updated `program` tuple. If the join with the `StageLattice` produces more than one tuple, then the *StageScheduler* arbitrarily chooses one of the next stages to run.

The StageScheduler and all compilation stages (built-in or runtime-installed) are interconnected via the simple dataflow illustrated in Figure 3. P2 uses this dataflow to schedule all OverLog programs, including compiler stages and user programs. It consists of a C++ "demultiplexer" that routes tuples from its input (on the left) to individual event handlers listening for particular tuple names. Arrows leaving the Demux element in the figure contain the name of the tuple for which the four components to the right listen.

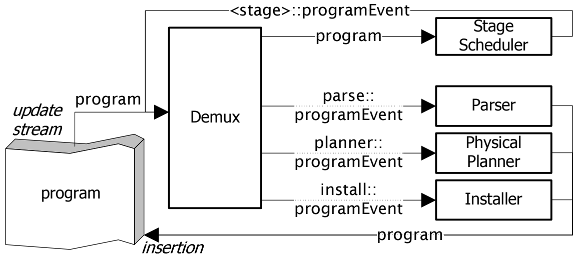

Figure 3: The cyclic dataflow of Evita Raced, showing only the default compilation stages.

Consider the simplicity of this approach as compared to the explicit stage-wiring sketched above. When a new compilation stage is installed at runtime, the Installer (Section 3.3.3) simply connects it to the *Demux* element, listening for `<stage>::programEvent` tuples, before updating the corresponding tuple in the `program` table. When the StageScheduler receives the updated `program` tuple, it uses the value of its $depends$ attribute to insert appropriate `StageLattice` tuples into the corresponding table of the system catalog. Subsequent `program` tuples will be redirected to the newly installed compiler stage (as `<stage>::programEvent` tuples) by the StageScheduler as the updated `StageLattice` dictates. Together, the StageScheduler and the Demux work much like an *eddy* operator [4]: they achieve flexible dataflow operator ordering via encapsulated routing decisions, rather than dataflow edges. In an eddy, this flexibility enables dynamic runtime reordering. In Evita Raced, it simplifies the installation of new compiler stages at runtime.

To sum up, the life 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 attribute to "Parser." The StageScheduler receives the `program` tuple, and uses the `StageLattice` table to decide how to proceed; in the case of the default stages in Figure 3, it routes a `planner::programEvent` to the Demux and eventually the Physical Planner, which goes around 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 (indicated by a non-null *depends* attribute in the `program` tuple), then after installation, the scheduler updates the `StageLattice`.

3.3 Compiler Bootstrapping

The previous architectural discussion neatly sidestepped a natural question: how is an Evita Raced compiler containing many OverLog stages bootstrapped, so that it can compile its own OverLog specification? As in many metaprogramming settings, this is done by writing a small bootstrap in a lower-level language. Evita Raced is initialized by a small C++ library that constructs the cyclic dataflow of Figure 3, including the three default stages shown, which are themselves written in C++. Together, this code is sufficient to compile simplified OverLog (local rules only, no optimizations) into operational P2 dataflows. We next describe each of these stages in a bit more detail, since they form the foundation of the Evita Raced runtime.

3.3.1 Parser

The Parser passes the program text it receives in the `programEvent` through a traditional lexer/parser library specified using flex and bison; 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 its position in the given program. By convention, the first term of a rule body is the event predicate of the rule, if one exists. By the same convention, the term in the last position for a rule is the head predicate.

3.3.2 Physical Planner

The Physical 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 positions of terms in the rule.

More specifically, for each rule the Planner considers each term (predicate, selection or assignment) in order of position attribute. The predicate representing the event stream is always planned first, and registers a listener in the Demux element (recall Figure 3). The terms following the event stream are translated, left-to-right, into a sequence of joins in the same way that the original P2 system did, so we do not address them further here.

We do mention three specific details. First, whereas the original P2 system translated a logical query plan directly to a software dataflow structure in C++, we chose to create an intermediate, textual representation of the dataflow, akin to Click's dataflow language, which can be examined and manipulated by programmers interested in modifying compiler output.

Second, unlike the original P2 system, we have introduced a number of access methods for in-memory tables. Our `predicate` relation contains the access method as one of the attributes, and we have modified the P2 physical planner to choose the appropriate dataflow element that implements the given access method.

Third, as mentioned before, OverLog rules may consist only of materialized predicates (e.g., "`table1 :- table2, table3.`"). An additional compiler stage written in OverLog converts such rules to (multiple) event rules via the semi-naïve evaluation *delta rewrite* of Loo et al. [24], e.g., "`table1 :- delta_table2, table3.`" and "`table1 :- table2, delta_table3.`". `delta_table` denotes a stream conveying insertions, deletions, or timeout refreshes to tuples of the table `table`. The delta rewrite compiler stage is written in OverLog using 9 rules (54 lines) and it is the first stage installed following compiler bootstrap.

3.3.3 Plan Installer

Given the output of the Physical Planner in the dataflow specification language, what remains is to parse that text, and construct the corresponding dataflow graph of C++ elements. We have implemented this "physical plan compiler" in C++, and housed it within the Installer stage. Once these elements and their connections are instantiated, the Installer stage connects them to the Demux.

3.4 Discussion

The declarative metacompilation concept in Evita Raced naturally caused us to design an extensibility architecture based on data modeling and dataflow, rather than library loading and control flow (function calls). While rule-based approaches have been implemented before to make optimizers more easily extensible [10, 12, 14, 22, 27], the internal implementation of Evita Raced is unique in its economy of mechanism. We aggressively reuse the native dataflow infrastructure, which both executes optimization code, and orchestrates stages via precedence tables and the StageScheduler cycle. One benefit of this design is that the Evita Raced infrastructure itself adds very little code (and code maintenance overhead) to the P2 engine: beyond the StageScheduler and the three bootstrap stages, no additional extensibility code was added to P2 to support Evita Raced. A second benefit is that even a major addition to the Evita Raced compiler entails 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. We return to these points in Section 6.

4. QUERY COMPILATION STAGES

Having described the Evita Raced infrastructure, we now turn our attention to the issue of specifying query optimizations in OverLog. In this section we describe three of the compiler stages we have developed for Evita Raced. Section 4.1 discusses a dynamic programming optimizer stage akin to that of System R [29] along with a modification to use a top-down search strategy akin to that of Cascades [30]. Section 4.2 describes a stage that performs the magic-sets rewrite on recursive OverLog programs [34]. Section 4.3 describes a protocol optimization specific to a wireless environment and reports on its benefit in a real sensor network setting. We conclude with a brief discussion of the compilation overhead added by the metacompiler in Section 4.4.

4.1 Query Optimization

The System R optimizer paper by Selinger, et al. is the canonical textbook framework for database query optimization [29]. The paper first laid out 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; OverLog is a fitting language to collect these statistics, especially in a distributed fashion over all relation partitions. In this section we describe a fairly faithful implementation of the Selinger paper written in OverLog. In Section 4.1.3 we extend this description with better selectivity estimation techniques using histograms. Finally, Section 4.1.4 presents a description of an optimizer stage that employs a top-down search strategy.

We focus first on the basic dynamic programming (DP) algorithm for the state-space enumeration at the heart of the System R optimizer, including the standard features for handling multiple access and join methods, and the "interesting orders" in subplans (which could be naturally generalized to other physical properties). This algorithm enumerates query plans for increasingly-large subgoals of the query optimizer. The task of the algorithm is to fill in a DP table with the lowest-estimated-cost query plan among all plans producing an *equivalent* output relation (i.e., plans composed of the same query terms and physical properties). In the System R optimizer, the *principle of optimality* is assumed to hold: the lowest-cost solution to some plan will be built from the optimal solutions

```
pg2 plan(@A, Pid, Rid, f_idgen(), PlanID, "Predicate", PredID,
          Plan, Schema, Card, Cost,
          OuterPos+1, AM, null, Sort) :-
        bestPlanUpdate(@A, Pid, Rid, PlanID),
        plan(@A, Pid, Rid, PlanID, _, _, _, OuterPlan,
            OuterSchema, OuterCard, OuterCost, OuterPos,
            _, _),
        rule(@A, Rid, Pid, _, _, _, _, TermC, _),
        predicate(@A, PredID, Rid, _, _, Tid, _,
            PredSchema, PredPos, _, _),
        PredPos < TermC,
        table(@A, Tid, _, _, _, _, TCard, _),
        f_contains(PredID, OuterPlan) == false,
        Card := OuterCard * TCard / 10,
        Cost := OuterCost + (OuterCard * TCard),
        AM := f_cons("SCAN", null),
        Plan := f_cons(PredID, OuterPlan),
        Schema := f_merge(OuterSchema, PredSchema),
        Sort := null.

pgn planUpdate(@A, Pid, Rid, PlanID, SubPlanID, Sort) :-
        plan(@A, Pid, Rid, PlanID, SubPlanID, _, _,
            _, _, _, _, _, _, Sort).
```

Figure 4: Scan (nested loop) join method.

to subplans. Thus dynamic programming can proceed in "bottom-up" fashion. The process is driven by having each rule contain an event predicate that listens for the generation of new subplans of k terms. For a given rule, the optimizer generates plans of size $k + 1$ terms by appending a single, as yet unused term from the rule body to an optimal plan of size k terms.

We first describe the rules for plan generation and conclude with the rules for optimal plan selection.

4.1.1 Plan Generation

The plan table stores query plans for each rule in the program. Among other things, it defines the physical properties (i.e., access methods, sort order, cost) associated with the plan and the set of terms (i.e., table predicates, selections, and assignments) that participate in the plan. The optimization begins when a tuple on the programEvent event stream is received. When such a tuple is received it is joined with the rule table to get all rules in the program, followed by the predicate table to obtain all table predicates in the rule. From this join result, a plan tuple is formed for each rule containing the rule's streaming predicate. This initial plan tuple seeds the bottom-up search strategy for each rule.

The optimizer is defined by a set of plan generation rules that extend the best k-term plan with a new thus far unused term from the rule body; examples appear in Figures 4 and 5. Each such rule joins the bestPlanUpdate event predicate (generated when a new k-term plan is found) with unused terms in the rule. If the new term considered is a predicate, then the new plan must define a join method that connects the optimal subplan and the predicate table via a physical join operator. The join methods we presently support are scanned and index-based nested-loop-joins, as well as merge-join. The rules for plan generation from rule predicates are defined around the supported join methods. Due to space constraints, we only show the rules that generate plans for nested-loop-join and index nested-loop-join access methods.

A nested-loop-join plan is generated for any table predicate appearing in the rule body. Rule pg2 in Figure 4 generates a (scanned) nested-loop-join plan on all rule body table predicates not mentioned in the OuterPlan variable of the plan predicate representing the subplan. The plan tuple representing the subplan is joined with the predicate table (to get all predicate terms in the rule body) followed by another join with the table table (to get each table's statistics). The result of these join operations produce all term predicates mentioned in the rule that have a matching table identifier definition. The selection predicate PredPos < TermC ensures that we do not join with the rule's head predicate (which is last in the rule's terms, by convention). The function f_contains tests for containment of the predicate identifier (PredID) in the subplan (OuterPlan). Any tuples that meet the constraints imposed by this rule generate a new plan tuple with the "SCAN" access method (since the predicate table will be scanned for each outer tuple). Each new plan tuple is given a new Plan variable that appends the PredID to the OuterPlan subplan variable . The cardinality (Card) and cost (Cost) estimates are given values based on the simple costing measures suggested by System R in the absence of indices; clearly these expressions can be enhanced, as we discuss in Section 4.1.3.

```
pg3 plan(@A, ...)  :-
        ...
        table(@A, Tid, Tablename, _, _, _, TCard, Sort),
        index(@A,Iid,Tablename,Key,Type,Selectivity),
        f_contains(PredID, OuterPlan) == false,
        f_indexMatch(OuterSchema, PredSchema, Key),
        Card := OuterCard * (Selectivity * TCard),
        Cost := OuterCost + Card,
        AM := f_cons(Type, Iid),
        ...
```

Figure 5: Index join method (diff from Figure 4).

An index-nested-loop-join plan is generated by rule pg3 in Figure 5. The main difference between this rule and rule pg2 is the additional index predicate, which adds index definitions to the resulting table predicate tuples. (The common prefix of pg2 is omitted in Figure 5 to save space.) The function f_indexMatch tests if the index can be used to perform the join using attributes from the best plan schema (OuterSchema) and attributes from the predicate table (PredSchema). Any tuple results from this plan are assigned cardinality and cost estimates based on some cost function, which uses the additional index selectivity information given by the Selectivity variable defined by the index predicate. We also support range predicates in our index-nested-loop-joins but do not show the 3 relevant rules here.

A merge-join performs a join of a plan with a table predicate mentioned in the rule body along some sorting attribute. The tuple set from the outer plan and the predicate table must be ordered by the sorting variable. The output of a merge-join operation preserves the sorting attribute order. Therefore, the plan predicate generated by the merge-join rule includes the sorting attribute in the value of the Sort variable. We note that the Sort variable in the table predicate identifies the sorting attribute of the table. A null valued Sort variable, in either the outer relation plan predicate or the inner relation table predicate, means that the relation is unordered, and must be explicitly sorted prior to the merge-join operator. The cost of a merge-join operator incorporates the cost of explicit sorting of either relation as needed.

4.1.2 Best plan selection

Figure 6 shows two rules that select the best plan from a set of equivalent plans, in terms of the output result set and the ordering properties of the result set. The bestCostPlan predicate picks the plan with the minimum cost from the set of equivalent plans. This aggregation query groups along the program identifier, rule identifier, plan list, and sort keys. The function f_setequals tests whether the set of term identifiers in its two input plans are the same, regardless of the order. The inclusion of the sort attribute

```
bp1 bestCostPlan(@A, Pid, Rid, Plan1, Sort1, a_min<Cost>) :-
          planUpdate(@A, Pid, Rid, _, Plan1, Sort1),
          plan(@A, Pid, Rid, _, _, _, _,
                Plan2, _, _, Cost, _, _, Sort2, _),
          f_setequals(Plan1, Plan2), Sort1 == Sort2.

bp2 bestPlan(@A, Pid, Rid, PlanID, Plan2, Cost) :-
          bestCostPlan(@A, Pid, Rid, Plan1, Sort1, Cost),
          plan(@A, Pid, Rid, PlanID, _, _, _,
                Plan2, _, _, Cost, _, _, Sort2, _),
          f_setequals(Plan1, Plan2), Sort1 == Sort2.

bp3 bestPlanUpdate(@A, Pid, Rid, PlanID) :-
          bestPlan(@A, Pid, Rid, PlanID, _, _).
```

Figure 6: Best plan selection.

in the group condition ensures the handling of what Selinger calls "interesting orders" [29], along with optimal subplans.

The aggregation rule `bp1` triggers whenever a new plan is added to the `plan` table (indicated by the `planUpdate` event). Then, the `bestCostPlan` predicate is used in rule `bp2` to select the identifier of the best plan, which is inserted into the `bestPlan` table. An update to the `bestPlan` table triggers a new `bestPlanUpdate` event that the plan generation rules, described in Section 4.1.1, use to build new candidate plans.

4.1.3 *Improving Selectivity Estimation*

For equality selection predicates, 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 equi-width 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

```
sys::ewhistogram::metadata(@LOCALHOST, "pred", 3, 10).
```

creates a 10 bucket equi-width histogram 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 attribute and 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. To exploit these histograms, the cost and selectivity estimation in the `plan` generation rules in Figures 4 and 5 can be modified to incorporate a join with the histogram data relation, and based on the bucket boundaries obtain density estimations for a given selection predicate.

4.1.4 *Top-down Optimization*

The bottom-up, dynamic-programming search strategy is a natural fit to a Datalog-based rule language. However, a top-down Cascades-style optimization strategy [30] is just as natural and intuitive in Evita Raced. We briefly describe how to implement the Cascades branch-and-bound algorithm in OverLog. The full optimization consists of 33 rules (204 lines).

In the Cascades optimizer, plans are classified into *groups*. A group is an equivalence class of plans that produce the same result.

```
c1 branch(@A, Pid, Rid, f_groupID(SubPreds), 0,
          SubPreds, Bound) :-
          winner(@A, Pid, Rid, GroupID, Cost),
          branch(@A, Pid, Rid, GroupID, Pos, Preds, Bound),
          Pos < f_size(Preds), Bound := Cost,
          SubPreds := f_removePredicate(Pos, Preds).
```

Figure 7: Branch and bound generation in top-down optimization.

The optimizer generates groups in a top-down order and within each group it searches for the cheapest plan, the *winner*. An upper bound is assigned to each group. A plan is pruned if its cost exceeds the group upper bound. The upper bound for a given group is initialized to the parent group upper bound (the root group upper bound is initialized to a cost of infinity) and continuously updated as new winner plans are discovered. The optimization terminates when the root plan group has fully explored or pruned all possible plans. The winner plan in the root group is then chosen to be the best-cost plan.

Figure 7 gives the single recursive rule that generates groups of plans in a top-down order (we omit the remaining rules due to space). A `branch` tuple contains the information that identifies a given group. Specifically, it identifies the group (`GroupID`), a branch position (`Pos`), the predicates (`Preds`) in the plan, and a bound (`Bound`). A separate `cost` relation maintains the cost of plans computed from the physical properties assigned to it during the optimization. A plan of size k is formed out of the current best-cost plan of size $k-1$ and the best-cost single predicate plan[4]. A new plan will only be generated if its cost is less than the bound value in the branch tuple containing the group to which the new plan belongs.

Suppose the initial query is $A \bowtie B \bowtie C$. The optimizer first initializes the `winner` relation with a tuple for each group (i.e., ABC, AB, AC, BC, A, B, C) each having a cost of `infinity`. A single rule seeds the recursive rule `c1` in Figure 7 by initializing the `branch` relation with a tuple that defines the root group (i.e., ABC) and starts the optimization at position 0 with the predicates (A, B, C) and a bound of `infinity`. The recursion proceeds in a top-down fashion. In the first step a new `branch` tuple is generated that removes the predicate in position 0 generating the group BC. Another rule will generate a branch group A at the same time. The recursion returns when a `winner` for group ABC has been discovered, which occurs when groups A and BC have been fully explored and the next branch position (e.g., 1) is set for group ABC. A group has been fully explored when its branch position reaches the end. As new winners are discovered the `Bound` variable is updated with the winning cost before proceeding to the next predicate.

4.2 Magic-Sets Rewrite

The magic-sets rewrite is an optimization that can reduce the amount of computation in recursive Datalog queries, via a generalization of basic "selection pushdown" ideas. It combines the benefits of top-down and bottom-up evaluation of logic [34].

Datalog-oriented systems like P2 perform a bottom-up (*forward chaining*) evaluation on each rule, starting with known facts (tuples), and recursively resolving body predicates to the head predicate. The advantage of this strategy is that the evaluation can be data driven (from known facts to possible deductions) in an optimizable set-oriented dataflow manner, and will not enter infinite loops for a class of statically verifiable *safe* programs. For example, the query in the shortest path program (Figure 1) asks for the

[4] The base case of this recursion costs single-table plans.

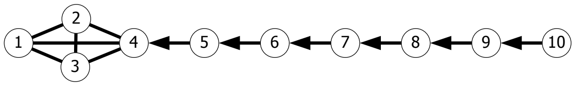

Figure 8: Experimental topology.

shortest path from all nodes to node `localhost:10000`. A bottom-up evaluation applies the `link` tuples to rule `r1`, creating initial `path` tuples. The program runs until it reaches a fixpoint. Any `shortestPath` tuples matching `localhost:10000` on their second attribute match the programmer's query. A deficiency of bottom-up evaluation is that it will generate some `path` and `shortestPath` tuples that do not have `localhost:10000` in the second attribute and therefore cannot satisfy the programmer's query. These irrelevant deductions are avoided in top-down evaluation.

Magic-sets rewriting adds extra selection predicates to the rules of a program to avoid the generation of irrelevant deductions. Conceptually, given a rule of the form

$$H_p \text{ :- } G_1, G_2, ..., G_k.$$

where H_p is the head predicate and $G_1 \ldots G_k$ are the goal predicates in the order of appearance in the rule, a magic-sets algorithm intersperses selection predicates $s_1, \ldots s_k$ to generate the rule

$$H_p \text{ :- } s_1, G_1, s_2, G_2, ..., s_k, G_k.$$

Facts for the new selection predicates are generated according to bindings of the variables of H_p in the user's query (e.g., `localhost:10000`), or other identified attribute bindings in the program. Appendix A gives further details on the magic-sets algorithm and describes the OverLog rules that perform the main steps involved in performing the rewrite.

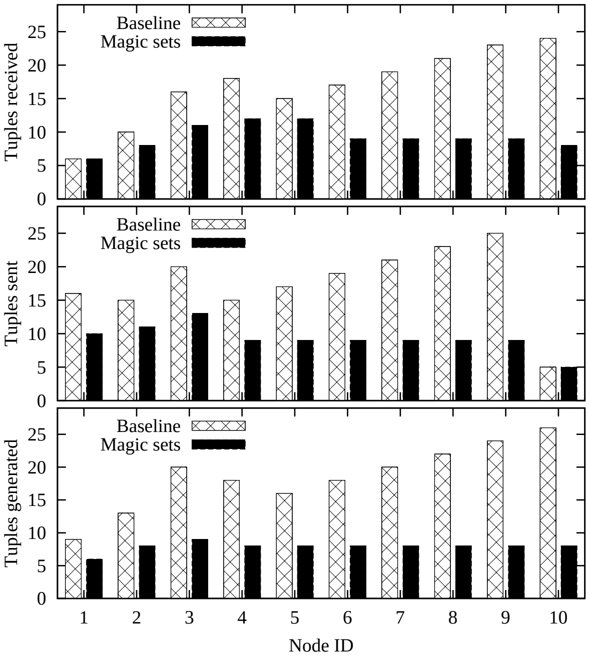

Figure 9: For each node (node ID on x axis), number of tuples received (top), sent (middle), and locally generated (bottom) on the y axis.

4.2.1 Magic Sets in the Network

To understand the effects of this rewrite, we describe two experimental runs of the shortest-path program in Figure 1, before and after the magic-sets rewrite. The two programs are executed in the simple link topology of Figure 8 (node 1 is given the address `localhost:10000`). Nodes are started up one at a time in order of the identifier, and the preloaded database 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).

The top of Figure 9 shows the number of tuples that each node receives from the network during shortest-path computation. The magic-sets rewritten program never causes more tuples to be received than the original, and results in increasingly fewer tuples received as we move to nodes farther away from the clique. That is because many paths that are generated in the original program to destinations within the clique other than node 1 are pruned early on and never transmitted all the way to the far end. Similarly, the middle plot in Figure 9 shows the number of tuples each node transmits. Again, the magic-rewritten program does a lot better. The inclusion of the magic-sets rewrite reduces the number of sends in all but one case (node 10). The node with identifier 10 is the only node with no incoming links and is therefore never burdened with network traffic other than its own; as a result, though its received tuple overhead benefits from magic sets, its transmitted tuple overhead is unaffected, since the node sends no extraneous path tuples other than its sole path towards node 1. Finally, `path` storage overhead is reduced by magic sets everywhere (Figure 9 bottom), since the rewrite prunes away irrelevant `path` tuples both received from the network, and generated locally.

4.3 Wireless Protocol Optimization

The previous two examples were well developed, traditional optimizations from the database literature that have applications in the networking domain. We now turn to an example taken directly from networking: optimizing the way in which tuples are sent from a given source to many destinations in a wireless environment. In such settings, a sender often has the choice to either communicate one-to-one directly with each desired receiver (*unicast* communication), or to communicate one-to-all with every node (*broadcast* communication). For unicast, the send and receive costs are proportional to the number of *intended* destinations, which could be lower than the size of the entire network. For broadcast, the send cost is the cost to send a single tuple. However, the receive cost is proportional to the number of nodes in the network, since every node will receive the tuple, even those that are not an intended destination.

The decision of when to use unicast, broadcast, or even *multicast*—one-to-some instead of one-to-all—can dramatically impact the design of a system, wireless or wired. For instance, much research in replicated systems concentrates on when to use each mode of communication to maximize throughput and minimize response times, and how to architect entire protocols around that decision [8, 19]. Moreover, with battery-powered networks such as wireless sensornets, radio send and receive operations dominate energy consumption, so minimizing communication cost is a primary concern [16]. The best choice depends not only upon the number of intended destinations (which is application-dependent) but also upon the relative costs of transmission and reception (which is dependent on the radio technology used). It is difficult and unusual for programmers to consider these factors jointly, so this is an ex-

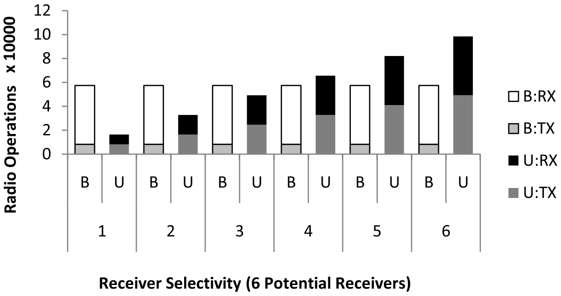

Figure 10: Radio operations (y axis) versus receiver selectivity of remote events (x axis), for a network of 7 nodes, broken down by Unicast, Broadcast, Transmit (TX) and Receive (RX).

cellent target for automatic optimization.

Evita Raced allows us to express this optimization in a relatively straightforward fashion. Consider the following rule template:

```
predRemote(@A,...):-
         predLocal_1(@B,...),...,predLocal_n(@B,...),
         predLink(@B, A).
```

The results of the above rule are to be sent from source node `B` to destination node `A`. For ease of exposition, we assume that the destination node `A` is determined by `predLink`, though generally it could be determined in any of the `predLocal` predicates.

The main idea of the rewrite is to gather selectivity information on intended destinations and use it to generate either a unicast or a broadcast version of the rule. The rewrite rule

```
destcountRemote(@B,a_countdistinct<A>,...) :-
         predLocal_1(@B,...),...,predLocal_n(@B,...),
         predLink(@B, A).
```

computes the number of distinct destinations for each unique `predRemote` tuple. This count is fed into a cost function of the general form

```
sendRemote(@B,...,U_Cost, B_Cost) :-
         destcountRemote(@B,Destinations,...),
         sys_network(@B, NetSize, SendCost, RecvCost),
         U_Cost := Destinations * (SendCost + RecvCost),
         B_Cost := SendCost + NetSize * RecvCost).
```

If `B_Cost < U_Cost`, then the original rule is rewritten as a broadcast rule, using a special broadcast address from the `wirelessBroadcast` relation that is uniquely defined in a wireless environment:

```
predRemote_broadcast(@BROADCAST,a_mkset<A>,...) :-
         predLocal_1(@B,...), ..., predLocal_n(@B,...),
         predLink(@B, A),
         wirelessBroadcast(@B, BROADCAST).
```

The resulting tuple contains the set of all intended destinations (collected by the `a_mkset` aggregation function), and the attribute values of the original tuple. Another rule that receives a tuple of type `predRemote_broadcast` will check if it is an intended destination (through a set membership check) and, if so, project the received tuple onto `predRemote`. On the other hand, if `B_Cost` $\geq$ `U_Cost`, then we simply revert to the original unicast version of the rule.

We implemented this rewrite in 8 rules (63 lines) for wireless sensornets, using Evita Raced as the compiler front end and DSN [6] as the back end and runtime. For testing, we fed as input to the compiler a simple event detection program that periodically sends alerts from one node to some fraction of the other nodes. This fraction can vary due to event types and node interest. Figure 10 shows the results of running this program on our seven-node sensornet testbed through 8200 events. Unicast is preferrable if the selectivity is below 4 nodes whereas broadcast is preferrable if the selectivity is at least 4 nodes. As selectivity increases and more nodes are interested in each event, the cost of unicast increases whereas the cost of broadcast remains constant. Selecting the best protocol leads to as much as a $3.5\times$ reduction in total radio operations, which in the sensornet case, translates into a nearly identical increase in energy savings and node lifetime.

While we have presented this broadcast vs. unicast rewrite as a static compile-time optimization, we have also implemented three additional compiler rules that enable a protocol to make the choice dynamically during runtime. The key difference is that the dynamic case includes the cost function and both the broadcast and unicast versions of the original rule in the rewrite's output.

4.4 Compilation Overhead

The metacompiler adds some overhead to the compilation process that did not exist in P2's original monolithic compiler. Even so, the most complex of our OverLog programs compile on the order of seconds with both the System R optimization and Magic-sets rewrite installed. For instance, our version of the Chord overlay written in OverLog [25] compiles in 31.607 seconds. As a comparison we took MIT's C++ Chord code and ran it through g++, which took 43.694 seconds. In the Declarative Networking setting, compilation in less than a minute is usually more than adequate, since networking programs typically define the ongoing behavior of a continuously-running, long-lived dataflow.

However, in some cases we have been sensitive to startup overheads in our use of P2. For instance, some of our performance experiments assume that the start times for all nodes will be identical. Given compilation times with a mean on the order of seconds, the variance of compilation time across nodes can lead to an unacceptable jitter in the start of the experiment. Compilation costs at install time can be reduced by precompiling an OverLog program into one of two possible intermediate representations. One available representation is rewritten OverLog, representing the original program after being processed by some number of OverLog-to-OverLog compilation stages. At installation time, Evita Raced can be configured to avoid any stages that have already been run. An alternative intermediate representation is the textual dataflow description generated by the Physical Planner. The load time of this representation is almost immediate, and even outpaces the installation time in the original monolithic compiler.

5. RELATED WORK

The pioneering work on extensible query optimizer architectures was done in the EXODUS [10] and Starburst [22, 27] systems, which provided custom rule languages for specifying plan transformations. The EXODUS optimizer generator used a forward-chaining rule language to iteratively transform existing query plans into new ones. Follow-on work (Volcano [14] and Cascades [12]) 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 [27]. The cost-based optimizer in Starburst was more declarative, taking a grammar-based approach to specifying legal plans and subplans [22]. These rule-based optimizers have been extremely influential in industry: the Microsoft SQL Server and Tandem optimizers are based on the Cascades design, and the optimizer in IBM's DB2 is based on the Starburst design.

The Starburst cost-based optimizer is the closest analog to Evita

Raced in the literature. In the paper, the authors explicitly draw analogies between their approach and Datalog, but stop short and distinguish between an optimizer – which in their view manipulates query plans – and a datalog engine, which manipulates data [22].

Relative to the previous rule-based extensible optimizers, Evita Raced innovates on a number of fronts. First, by treating query plans as data, it can reuse the dataflow engine to implement the optimizer rule evaluation. This metacompilation approach provides significant *economy of mechanism*, a well-established principle in developing reliable software [28]. Second, Evita Raced allows all aspects of the optimizer to be extended, including not only the plan transformations and cost functions, but also the search strategies. For example, we switched from a System-R-based, bottom-up optimizer to a Cascades-based, top-down optimizer in response to a reviewer's request with great ease in fewer than 24 hours (Section 4.1.4); we are not aware of any other optimizer framework where this kind of radical change would be so easy to achieve. Third, the tabularization of code in Evita Raced enables code analysis via standard queries – for example, stratification tests or other static program analyses useful for debugging. Finally, the use of a logic-based language brings a measure of semantic rigor to the extensibility language, as well as theoretical tests from the Datalog literature that help in program understanding. This potentially alleviates some traditional software engineering problems with understanding rule interactions in more ad-hoc rule languages.

Another interesting extensible query optimizer is Opt++ [17], an elegant object-oriented design for an optimizer that is customizable via inheritance and overloading. 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 much higher-level declarative programming framework.

6. DISCUSSION

When we began this project, we did not know whether a fully declarative compiler was feasible or useful. We have been surprised by just how positive our experience with Evita Raced has been. It has allowed us to upgrade P2 from having essentially no optimizations of note, to having quite a sophisticated suite of optimizations, rewrites, and program analyses. However, not everything was smooth, and in this section we list some of the chief lessons from our experience, which suggest research directions for improving this line of work.

The most difficult problems we faced were due to discrepancies between P2's runtime behavior and Datalog semantics. In fact, the current state of the system and language as described in Section 2.1 is already quite a bit crisper than what we had when we started this work. Along the way, we often had to reason about operational issues at the dataflow execution level, and find work-arounds that ended up complicating our optimization code. Since then, the P2 group has moved the runtime onto a much cleaner semantic footing, and today our optimization rules no longer require work-arounds for the remaining implementation flaws mentioned at the end of Section 2.1. This experience has strongly reinforced our belief that truly declarative languages with clean semantics are superior to more ad-hoc event-condition-action rule languages. Coding and debugging is significantly eased by the ability to ignore runtime considerations, and instead work from declarative semantics.

The second class of problems we faced came from our roots in Datalog, and the challenge of scaling the cognitive burden of Datalog-style programming into hundreds of rules developed by multiple people. The downside of OverLog's conciseness is that even smallish batches of rules contain significant conceptual density. We found that understanding non-trivial amounts of OverLog – especially someone else's OverLog – can be difficult simply because the intent of the program gets complex quickly. This is exacerbated by the notation of variable unification, which relies on the programmer being able to visually match variable names across terms in a rule. Additionally, the need to pay close attention to the position of variables in comma-separated lists is vexing when table arities get above 3 or 4. Finally, the lack of modularity or encapsulation in Datalog syntax does nothing to encourage good code structuring and reuse, topics that matter a lot when the number of rules climbs into the many dozens. These latter problems can be addressed via syntactic reworkings and extensions of OverLog, an interesting design problem we are actively considering.

A third class of problems arises from the semantics of OverLog's extensions to Datalog, especially with respect to event tables. While OverLog ostensibly promises Datalog-like semantics within a (local) fixpoint computation, in order to truly understand the behavior of a long-running OverLog program, you have to understand what happens across multiple fixpoints – i.e., across the handling of multiple event tuples. That means thinking declaratively within a fixpoint, but reasoning about ordering among events that are handled across multiple fixpoints. It is not clear how to address this; one possible direction is to bring concepts from temporal logic into OverLog to reason about this more declaratively.

Finally, in talking to colleagues in industry, one constant we hear is that – regardless of the underlying extensibility architecture – the development and maintenance of query optimizers is a major challenge. For one thing, it is hard to debug code when the output's correctness (e.g., minimality of cost) is too expensive to verify in general. Also, optimizers simply contain a lot of logic, including statistics, search algorithms, and manipulation of complex data structures with a lot of object sharing (e.g., of subplans). Our experience with Evita Raced is that declarative programming and relational modeling can help mitigate these challenges quite a lot, but there is no panacea – good design and taste are still required to successfully separate concerns in the problem space (e.g., measurement vs. modeling in statistics generation, logical vs. physical query plan issues, etc.), and develop well modularized solutions.

7. CONCLUSION AND FUTURE WORK

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 the previous section, 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 are now at a point where we can 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 traditional 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 of intermediate program forms; language wrappers for secure networking functionality in the manner of SecLog [1]; stratification detectors and other static code analyses. None of these are performance optimizations per se, but all fit well within an extensible, declarative program manipulation framework. As OverLog and P2 mature, we expect the use of the metacompilation approach to get even easier, and expect it will (recursively) help us to implement better versions of the language and runtime. 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.

Acknowledgments

Thanks to Goetz Graefe and Hamid Pirahesh for helpful insights and perspective, and to Kuang Chen for editorial feedback.

8. REFERENCES

[1] M. Abadi and B. T. Loo. Towards a Declarative Language and System for Secure Networking. In *International Workshop on Networking Meets Databases (NetDB)*, 2007.

[2] T. Anderson, L. Peterson, S. Shenker, and J. T. (Eds). Report of nsf workshop on overcoming barriers to disruptive innovation in networking. Technical Report 05-02, GENI Design Document, Jan. 2005.

[3] M. P. Ashley-Rollman, M. De Rosa, S. S. Srinivasa, P. Pillai, S. C. Goldstein, and J. D. Campbell. Declarative Programming for Modular Robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications*, 2007.

[4] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[5] N. Belarmani, M. Dahlin, A. Nayate, and J. Zheng. Making Replication Simple with Ursa. In *SOSP Poster Session*, 2007.

[6] D. Chu, L. Popa, A. Tavakoli, J. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The Design and Implementation of a Declarative Sensor Network System. In *SenSys*, 2007.

[7] T. Condie, J. M. Hellerstein, P. Maniatis, and S. R. T. Roscoe. Finally, a use for componentized transport protocols. In *HotNets IV*, 2005.

[8] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *OSDI*, 2006.

[9] J. Eisner, E. Goldlust, and N. A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proc. Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT-EMNLP)*, 2005.

[10] D. D. G. Graefe. The EXODUS Optimizer Generator. In *SIGMOD*, 1987.

[11] P. Grace, D. Hughes, B. Porter, G. Blair, G. Coulson, and F. Taiani. Experiences with Open Overlays: A Middleware Approach to Network Heterogeneity. In *EuroSys*, 2008.

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

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

[14] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, 1993.

[15] J. M. Hellerstein. Toward network data independence. *SIGMOD Rec.*, 32(3), 2003.

[16] J. L. Hill and D. E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.

[17] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3), 2000.

[19] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *SOSP*, 2007.

[20] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-Sensitive Program Analysis as Database Queries. In *PODS*, 2005.

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

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

[23] B. T. Loo. *The Design and Implementation of Declarative Networks*. PhD thesis, University of California, Berkeley, 2006.

[24] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*, 2006.

[25] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[26] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*, 2005.

[27] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule-Based Query Rewrite Optimization in Starburst. In *SIGMOD*, 1992.

[28] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), Sept. 1975.

[29] P. G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 1979.

[30] L. D. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H. min Wu, and B. Vance. Exploiting upper and lower bounds in top-down query optimization. In *International Database Engineering and Application Symposium*, pages 20–33, 2001.

[31] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.

[32] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT Protocols Under Fire. In *NSDI*, 2008.

[33] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Distributed monitoring and forensics in overlay networks. In *EuroSys*, 2006.

[34] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.

[35] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *SIGMOD*, 2007.

```
materialize(sup,infinity,infinity,keys(2,3,4)).
materialize(adornment,infinity,infinity,keys(2,5,6)).
materialize(idbPredicate,infinity,infinity,keys(2,3)).

mg1 goalCount(@A, Pid, Name, a_count<*>) :-
          idbPredicate(@A, Pid, Name),
          adornment(@A, Pid, Rid, Pos, Name, Sig).

mg2 magicPred(@A, Pid, Name, Sig) :-
          goalCount(@A, Pid, Name, Count),
          adornment(@A, Pid, _, _, Name, Sig).
          Count == 1.

mg3 sup(@A, Pid, Rid, Pos, Name, Schema) :-
          magicPred(@A, Pid, Name, Sig),
          rule(@A, Rid, Pid, _, HeadPid, _, _, _),
          predicate(@A, HeadPid, Rid, _, Name, _, _, Schema,
                    _, _, _),
          Schema := f_project(Sig, Schema),
          Name := "magic_" + Name, Pos := 0.

mg4 supNext(@A, Pid, Rid, Pos+1, Schema) :-
          sup(@A, Pid, Rid, Pos, Name, Schema).

mg5 sup(@A, Pid, Rid, Pos, Name, Schema) :-
          supNext(@A, Pid, Rid, Pos, PrevSupSchema),
          rule(@A, Rid, Pid, RuleName, _, _, _, _),
          predicate(@A, _, Rid, _, _, _, _, Schema, Pos, _, _),
          Name := "sup_" + RuleName + "_" + f_tostr(Pos),
          Schema := f_merge(PrevSupSchema, PredSchema).

mg6 adornment(@A, Pid, Rid, Pos, Name, Sig) :-
          supNext(@A, Pid, Rid, Pos, PrevSupSchema),
          idbPredicate(@A, Pid, Name),
          rule(@A, Rid, Pid, _, _, _, _, _),
          predicate(@A, _, Rid, _, Name, _, _,
                    Schema, Pos, _, _),
          Sig := f_adornment(PrevSupSchema, Schema).
```

Figure 11: Rule/Goal graph traversal rules.

APPENDIX

A. MAGIC-SETS RULE DESCRIPTION

Ullman's textbook description of magic sets [34] can be viewed as a traversal of a directed graph called the *Rule/Goal* graph. We briefly review his description here as a refresher to help clarify the declarative specification that follows. For a more thorough introduction to the algorithm, we direct the reader to the textbook [34]. The vertices of the Rule/Goal graph are rules and goals, and the edges represent data dependencies. Briefly put, a goal points to a rule if it appears in the rule body, while a rule points to a goal if that goal appears in the rule head. In the magic-sets algorithm, the *Rule/Goal* graph is rooted by the query predicate. The traversal of the *Rule/Goal* graph generates new *magic* predicates that contain the set of variable bindings presented in a program's derived predicates. A magic predicate is generated for each "goal" vertex that defines a unique "adornment", where an adornment is a variable-binding pattern that indicates which variables are free and which are bound to a constant. A *supplementary* predicate is also created for all encountered "rule" vertices during this graph traversal. Supplementary predicates capture the way variable bindings can be passed "sideways" from left-to-right through the terms of a rule body.

To give a flavor of the OverLog implementation of magic-sets, Figure 11 shows six rules that create the magic and supplementary predicates through a traversal of the *Rule/Goal* graph (rules in the graph correspond to the `rule` predicate, and goals are given by the `predicate` predicate). These six rules correspond to steps i and ii of Algorithm 13.1 in Ullman's textbook [34, Chapter 13].

The `adornment` predicate contains the predicate name (`Name`) and an adornment string (`Sig`), which is initially populated (by a single rule, not shown) with the query predicate adornments. Rule `mg1` counts the number of adornments for each *IDB* predicate. If this count is unique (`Count == 1`) in rule `mg2`, then a `magicPred` tuple is created. Rule `mg3` triggers on a `magicPred` tuple and, for each rule whose head predicate is named by the `magicPred` tuple, it generates a `sup` predicate with a `Schema` attribute containing the bound variables that exist at the given rule position. Rule `mg4` detects a new `sup` predicate (like the one generated for the rule head) and triggers an event for the subsequent `sup` predicate position in the given rule. The three way join in rule `mg5` produces a tuple that contains the schema of the previous `sup` predicate (`PrevSupSchema`) and the schema of the predicate (`Schema`) in the subsequent rule position, should one exist. Two more rules (not shown) move the `supNext` position forward if the given rule position does not identify a predicate. The head `sup` predicate schema in rule `mg5` contains all the variables from the previous `sup` predicate and the schema of the current predicate, since this schema represents the bound variables that will exist in the subsequent rule position. Rule `mg6` creates an `adornment` out of the predicate in the given rule position, if that predicate is part of the *IDB*. The `f_adornment` function creates a new signature from the bound variables in the `PrevSupSchema` attribute, and the variables in the predicate `Schema` attribute. At the end of the rule/goal graph traversal, those predicates that define a unique adornment become magic predicates, and the rules that mention these magic predicates are rewritten using the information contained in the `sup` table.