Evita Raced: Metacompilation for Declarative Networks

Under Submission. Please do not redistribute

Tyson Condie
UC Berkeley

Joseph M. Hellerstein
UC Berkeley

Petros Maniatis
Intel Research Berkeley

ABSTRACT

There has been renewed interest in recent years in applying declarative languages to new application domains outside of traditional data management. Since these are relatively early research efforts, it is important that the architectures of these new 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 networking engine. Specifically, we describe our design and implementation of *Evita Raced*, an extensible query optimizer for the Overlog language used by the P2 declarative networking system. Evita Raced is a *metacompiler*: an Overlog compiler written in Overlog. We describe the architecture of Evita Raced, including its extensibility interfaces and its dataflow-based runtime structure. We demonstrate that a declarative language like Overlog is well-suited to expressing traditional and novel query optimizations, 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 languages to a variety of applications outside the traditional boundaries of data management. Examples include work on compilers [15], computer games [28], security protocols [16], and modular robotics [4]. Among the most active of these new areas is the work on *Declarative Networking* initiated by the P2 project [18, 25], and followed on by variety of recent related efforts [3, 5, 7, 26]. The initial motivation for Declarative Networking came from the recursive graph-traversals in various network routing protocols, which are natural to express in recursive query languages like Datalog [20]. Subsequently, the argument for the fitness of declarative languages to networking was expanded to include the asynchronous message-handling inherent in network protocols: the case was made that message-handling is compactly expressed as joins of data streams of messages with persistent though transient "rendezvous" or "session" state [18, 19]. These intuitions were made concrete via working implementations of complex network protocols at various levels of the protocol stack, often with drastically reduced code sizes that were expressed in an intuitive manner quite similar to protocol inventors' pseudocode [7, 19].

Declarative Networking and related topics have 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 had received minimal attention in recent years. Yet surprisingly, the proponents of Declarative Networking have not to date applied their ideas to the design of their own systems: the P2 declarative overlay system is implemented in C++ [19], and the DSN declarative sensor network system is implemented in an embedded dialect of C [7]. This is particularly disappointing given the still-emerging use cases for these systems, which are likely to require the systems to be adjusted to unforeseen requirements.

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 optimizer of P2 as a *metacompiler*: a compiler (optimizer) for the P2 language, Overlog, that is itself written in Overlog. We call 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. For example, our implementation of the traditional System R dynamic programming algorithm comprises only 38 Overlog rules (225 lines of code); our implementation of the Magic Sets Rewriting optimization for recursive queries is not only compact (68 rules, 264 lines), but also nearly a direct translation of the description from Ullman's course notes on the subject [27]. 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 approach in much the same way as networking protocols. Even non-recursive optimization logic – such as parts of Ullman's magic-sets algorithm – 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. Second, we show that a variety of traditional and novel query optimizations are easy to express in a recursive, declarative language. Finally, we evaluate the simplicity and applicability of our design via a full-fledged im-

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

plementation of an Overlog query optimizer for the P2 Declarative Networking engine, which also cross-compiles code that runs on the DSN wireless sensor network platform [7]. We also include a discussion of the various compromises that we had to make along the way in the interest of pragmatism. Despite the various engineering details that arose, we believe that declarative metacompilation is a clean, architecturally parsimonious way to build the next generation of extensible query optimizers, particularly as declarative languages get applied in new domains where the relevant optimizations remain unknown.

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 query engine, and, by default, participates equally in every "query." In parallel programming terms, P2 supports the Single-Program-Multiple-Data (SPMD) model of parallel computation.

The P2 runtime at each node consists of a compiler—which parses programs, optimizes them (minimally), and physically plans them—a dataflow executor, and access methods. The P2 compiler is monolithic and implemented in an imperative language (C++). The subject of this work is the replacement of this monolithic compiler with a runtime-extensible compiler framework that admits not only imperative but also declarative optimizers and other compilation stages. 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

The original paper on P2 presented Overlog in an ad-hoc manner as an event-driven language. Since that time, the P2 group has refined the Overlog language and the P2 runtime semantics a fair bit. However, the only published discussion of the current state of Overlog is in the tutorial document included with the P2 source distribution, which is also quite informal.

In order to build a new optimizer for P2, we had to work with the P2 developer group and the codebase to get familiar with the semantics of Overlog as it currently stands. In this Section we overview those semantics, since they form the setting for our work; we did not attempt to extend or clean up the language and its semantics in this project (see Section 6 for some unresolved problems we encountered).

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 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* [2]. An example Overlog program that we will use throughout is shown in Figure 1. Overlog introduces some notable extensions to Datalog:

Horizontal partitioning—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 specify 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 [18].

```
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).
```

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

Soft State and Events—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. At any instant in time, 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 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 for a single P2 node at a snapshot in time. P2 has no defined semantics across time and space (in the network); we describe the relevant operational semantics of the prototype in Section 2.2.2.

Deletions and Updates—Overlog, like SQL, supports declarative expressions that identify tuples to be deleted in a deferred manner after query execution completes. To this end, any Overlog rule in a program can be prefaced by the keyword `delete`. 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

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

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 the replacement is separated into an insertion and a deletion: the deduction of the new fact to be inserted is visible within the current fixpoint, whereas the deletion of the original fact is deferred until after the fixpoint is computed.

2.1.1 A Canonical Example

To illustrate the specifics of Overlog, we briefly revisit a shortest paths example (Figure 1), similar to that of [18], but with fully-realized Overlog syntax that runs in P2. The three `materialize` statements specify that `link`, `path` and `bestpath` 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 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 `Z`, the output of which is to be stored back at `X`. As described in the earlier work on P2 [18, 20] such rules can be easily rewritten so that the body predicates all have the same location specifier; the only communication then is shipping the results of the deduction to the head relation's location specifier. In Section 4.3 we briefly describe how we reimplemented this functionality in a *localization* compiler stage written in Overlog, within Evita Raced.

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 [14]. Like Click, the P2 runtime supports dataflow *elements* (or "operators") of two sorts: pull-based elements akin to database iterators [11], 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 [19].

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.

P2 currently has no 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 query predicates.

[3]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.

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 as discussed in [8]. However, the original P2 query optimizer always assembles these network elements in a fixed manner, and we did not choose to change that code in our work here. The basic pattern 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.

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, which is either an *insert* tuple (i.e., the result of a normal deduction) or a *delete* tuple (the result of a `delete` rule or a primary-key update). We will refer to this tuple as the *current tuple*.
2. The value of the system clock is noted in a variable we will call the *current time*. This is the time that will be used to determine the liveness of soft-state tuples.
3. The current tuple is, logically, appended to its table.
4. If the current tuple is an insert tuple, the dataflow corresponding to the Overlog program is initiated and run to a local fixpoint following traditional Datalog semantics, with the following exception: during processing, any non-local derived tuples are buffered in a *send queue*, as are any derived tuples to be deleted.
5. If, instead, the current tuple is a delete tuple, the dataflow is run to a local fixpoint, but newly-derived local tuples (including the current tuple) are copied to a *delete queue*, and newly-derived non-local tuples are marked as delete tuples before being placed in the send queue so as to cascade the deletions to remote nodes' databases.
6. All tuples in the delete queue are deleted from their associated tables, and the delete queue is emptied.
7. The send queue is flushed across the network, with any local updates inserted into the local input queue.

Some additional operational details we discovered in P2 are discussed further in Section 6.

3. DECLARATIVE OPTIMIZATION

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 via dataflow programs (query plans) running over a set of tables. 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 – 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. 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.

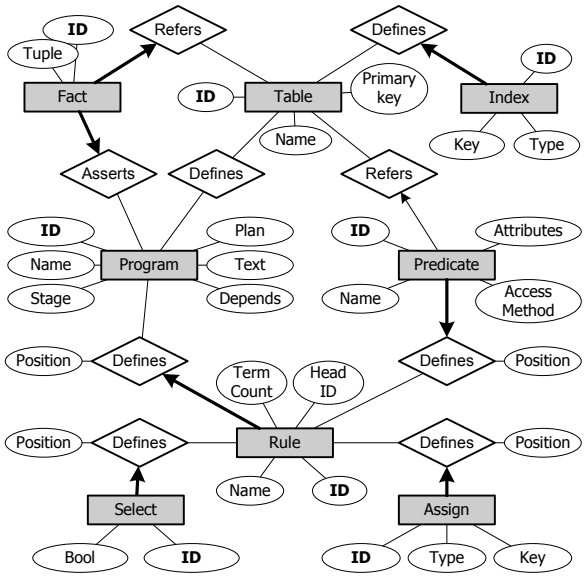

Figure 2: ER Diagram of a query plan in P2.

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 we developed that captures the properties of an Overlog program, and its associated P2 dataflow query plans. 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 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.

Having (after some design iterations) constructed the ER diagram, converting it to relational format was straightforward. Table 1 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 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.

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

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 dataflow program of one or more P2 elements in C++, and compile the result into the P2 binary; 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 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 stages. The $plan$ attribute pertains to the physical planner stage, which is described in Section 3.3.2. We also expose the dataflow registration facility of Evita Raced to programmers inserting arbitrary Overlog programs (not compiler stages) into a P2 node; for these generic 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 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. 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 that inserts a `program` tuple into the `program` ta-

ble to signal 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 compilation. 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* component 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, as described in the previous section. The *StageScheduler* is responsible for shepherding compilation stage execution according to the `StageLattice`. Given a `program` update, it checks the lattice to identify a next stage that can be invoked, and generates the `<stage>::programEvent` tuple that will start that stage; the contents of the tuple are the same as those of the updated `program` tuple.

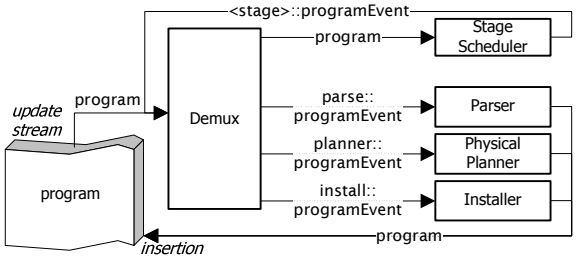

Figure 3: The cyclic dataflow of Evita Raced, showing only the default compilation stages.

The StageScheduler and any compilation stages (whether built-in or runtime-installed) are interconnected via the simple dataflow illustrated in Figure 3. This is the same dataflow that P2 constructs for all Overlog 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 (the arrows leaving the Demux element in the figure contain the name of the tuple for which the four components to the right listen). We have not changed P2's dataflow architecture [19], and its details are not required for the rest of this paper.

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. Then the StageScheduler who receives the updated `program` tuple uses the value of its $depends$ attribute to insert appropriate `StageLattice` tuples into the corresponding table of the system catalog. Subsequent `program` tuples being compiled will be directed to the newly installed compiler stage by the StageScheduler as the updated `StageLattice` dictates.

To sum up, the life cycle 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 routes a `planner::programEvent` to the Demux and eventually the Physical 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.

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 [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 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 C++ dataflow 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 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 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 have no event predicate (e.g., "`table1 :- table2, table3.`"). The physical planner converts such rules to (multiple) event rules via the *delta rewrite* of Loo, et al. [18]. (E.g., "`table1 :- delta_table2, table3.`" and "`table1 :- table2, delta_table3.`".) As in [18] `delta_table` denotes a stream conveying insertions, deletions, or timeout refreshes to tuples of the table `table`.

3.3.3 Plan Installer

Given the output of the Physical Planner in the dataflow specification language, what remains is to parse the textual representation of the dataflow, construct the corresponding C++ elements, and "wire them up" accordingly. 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 Plan Installer stage stitches them into the P2 runtime's overall dataflow graph, as described in [19]. As noted in 3.2, this infrastructure made it easy for us to extend P2 with the ability to modify its dataflow graph at runtime, a feature not available in the released system.

3.4 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 any rule-based system is easier to extend than a procedural system, the internal implementation of Evita Raced is especially elegant, 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 three bootstrap stages, no additional extensibility code was added to P2 to support Evita Raced.

Despite its simplicity, Evita Raced is flexible enough that we 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, we have been 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 in a manner similar to that of [3]. Second, we have been 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 [7].

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. Section 4.2 describes a stage that performs magic-sets rewrite on recursive Overlog programs. Finally, Section 4.3 describes our reimplementation in Evita Raced of localization from Loo et al. [18].

```
pg1 plan(@A, Pid, Rid, PlanID, SubPlanID, Type, TypeID,
         Plan, Schema, Card, Cost, Pos, AM, Sort, TermC) :-
        systemr::programEvent(@A, Pid, _, _, _, _, _),
        rule(@A, Rid, Pid, _, _, _, TermC),
        predicate(@A, PredID, Rid, _, _, _, _,
                      Schema, Pos, _, _),
        Pos == 1,
        PlanID := f_idgen(), SubPlanID := null,
        Type := "Predicate", TypeID := PredID,
        Plan := f_cons(PredID, null),
        Card := 1, Cost := 1,
        AM := "STREAM", Sort := null.
```

Figure 4: Plan seed rule.

4.1 System R Optimization

The System R optimizer paper by Selinger, et al. is the canonical textbook framework for database query optimization [24]. 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; Overlog is a fitting language to collect these statistics, especially in a distributed fashion over all relation partitions. In our description we omit statistics gathering.

We focus instead on the basic dynamic programming algorithm for the state-space enumeration at the heart of the System R optimizer. This algorithm enumerates query plans for increasingly-large subgoals of the query optimizer. It fills in a dynamic programming table keyed by the plan identifier. The dynamic programming task is to fill in this table with the lowest-estimated-cost query plan among all plans producing an *equivalent* output relation (i.e., plans composed of the same terms). 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 to subplans. Thus dynamic programming can proceed in "bottom-up" fashion. Each rule's event predicate is the "outermost," streaming relation in a query plan for the rule. For a given rule, the optimizer generates plans of size k terms by appending a single (thus unused) term from the rule body to the optimal plan of size $k-1$ terms.

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

4.1.1 Plan Generation

Figure 4 shows the optimizer rule that creates the initial plan for each rule from the event predicate. The `plan` tuple contains a query plan for a given rule, and the plan's `size` reflects the number of term identifiers in the $Plan$ attribute (i.e., the number of leaves in the plan tree). The optimizer listens on the `programEvent` event stream in rule `pg1`, which will start the optimization process. The `programEvent` tuple is subsequently joined with the `rule` table along the Pid (program identifier) attribute to obtain the set of rules defined in the input program. The resulting set of rule tuples are each joined with the `predicate` table along the Rid (rule identifier) attribute. The result of this join produces 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. For each event `predicate` tuple in each rule of the input program, rule

```
pg2 plan(@A, Pid, Rid, f_idgen(), PlanID, "Predicate", PredID,
         Plan, Schema, Card, Cost,
         OuterPos+1, AM, null, TermC) :-
       bestPlanUpdate(@A, Pid, Rid, PlanID),
       plan(@A, Pid, Rid, PlanID, _, _, _, OuterPlan,
           OuterSchema, OuterCard, OuterCost, OuterPos,
           _, _),
       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).

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

Figure 5: Scan-join access method.

pg1 adds a new plan to the plan table. The $Plan$ attribute represents the initial plan list, starting with the term identifier ($PredID$) that identifies streaming predicates. Plan generation proceeds by appending term identifiers to this list when constructing new plans from optimal subplans.

The optimizer defines a set of plan generation rules that extend the best plan found with k terms with a new thus far unused term from the rule body. 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 an access method that identifies the physical join operator, which takes the optimal subplan and "joins it" with the predicate table. The access methods we presently support are scanned and index-nested-loop-join, as well as merge-join. The rules for plan generation from rule predicates are defined around the supported access methods. Due to space constraints, we only show the rules that generate plans for nested-loop-join and index nested-loop-join access methods.[4]

A nested-loop-join plan access method is generated for any table predicate appearing in the rule body. Rule pg2 in Figure 5 generates a (scanned) nested-loop-join plan on all rule body table predicates not mentioned in the $OuterPlan$ attribute of the plan predicate representing the subplan. The plan tuple representing the subplan is joined with the predicate (all predicate terms in the rule body) table followed by another join with the table (all tables in the system) table. 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 consider the rule head predicate (last in the rule's terms by convention). The function $f\_contains$ tests for containment of the predicate in the subplan (outer plan). 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 $Plan$ attribute that appends the $PredID$ to the $OuterPlan$ subplan attribute. The cardinality ($Card$) and cost ($Cost$) estimates are given values based on some costing measure, which in this case makes use of best plan cardinality and cost estimates, and the predicate table cardinality.

An index-nested-loop-join plan is generated by rule pg3 in Figure 6. 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 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 $Selectivity$ index predicate attribute. We also support range predicates in our index-nested-loop-joins but do not show the 3 relevant rules here.

```
pg3 plan(@A, ...)  :-
          ...
          table(@A, Tid, Tablename, _, _, _, TCard, _),
          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 6: Index-join access method (diff from Figure 5).

```
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 7: Best plan selection.

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 attribute. 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 $Sort$ plan predicate. We note that the $Sort$ attribute in the table plan table identifies the sorting attribute of the table. A $null$ valued $Sort$ attribute, in either the outer relation plan predicate or the inner relation table predicate, means that the relation is unordered, and must be presorted prior to the merge-join operator. The cost of a merge-join operator depends on need to presort either relation.

4.1.2 Best plan selection

Figure 7 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 attribute. The function *f_setequals* tests whether the set of term identifiers in the two argument plans are the same, regardless of the order. The inclusion of the sort attribute in the group condition ensures the handling of what Selinger calls "interesting orders" [24], along with optimal subplans.

The aggregation rule bp1 triggers whenever a new plan is added

[4] A complete description of our rules in an extended technical report is available to the conference chair during double blind reviewing.

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.

4.2 Magic-Sets Rewrite

The magic-sets rewrite is an optimization that can reduce the amount of computation in recursive Datalog queries. It combines the benefits of top-down and bottom-up evaluation of logic, without the disadvantages of either [27].

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 is data driven (from known facts to possible deductions) and will not enter infinite loops for some statically verifiable *safe* programs. For example, consider again a snippet of the shortest path program, in Figure 8. The query asks for the shortest path from all nodes to node "localhost:10000". A bottom-up evaluation applies the `link` tuples to rule `r1`, creating initial `path` tuples. Those are `path` tuples until it reaches a fixed-point. Any `path` tuples matching "localhost:10000" on their second attribute match the programmer's query.

Bottom-up evaluation generates some `path` tuples that do not have "localhost:10000" in the second attribute and therefore cannot satisfy the programmer's query. In contrast, top-down (*backward chaining*) evaluation (e.g., in the Prolog language), starts with the query predicates 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. In the example, a top-down evaluation unifies the query predicate with the head predicate of rules `r1` and `r2` [5]. Therefore, the `path` predicate unification binds the $@X$ attribute to the current node identifier and the Y attribute to "localhost:10000" in both rules, which is carried over to the predicates in the rule body. The binding of $@X$ and Y attributes in rules `r1` and `r2` means that these rules will only look for tuples of the form $path(LOCALHOST, "localhost:10000", P, C)$ as well as tuples that can help form such `path` tuples, but nothing else [6].

Ullman's magic sets algorithm operates in three phases: program analysis, program rewrite, and filter population.

Magic-sets algorithms add extra selection predicates to the rules of a program to emulate the goal-oriented execution of top-down evaluation (this is sometimes called *sideways information passing* or SIP). Conceptually, given a rule of the form

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

where H_p is the head predicate p and $G_{1,...,k}$ are the goal predicates in the order of appearance in the rule, a magic-sets algorithm intersperses selection predicates $s_{1,...,k}$ to generate rule

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

[5] In P2 the location attribute of a query rule is always bound to the local node identifier.

[6] A detail of the SPMD networked nature of Evita Race is that queries run everywhere imply a binding for the location specifier attribute to the local host's address, conveyed by the `LOCALHOST` macro, hence the contents of query tuple.

```
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.

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

Figure 8: Relevant rules from Figure 1.

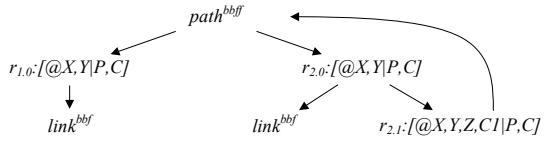

Figure 9: Rule/Goal graph of the program in Figure 8.

Facts for the magic predicates are generated according to bindings of the attributes of H_p in the user's query, or other identified attribute bindings in the program. In Evita Raced we create a magic-sets optimization that implements the description given in Ullman's course notes [27]. A very brief overview by example follows.

4.2.1 Magic-sets By Example

In the program analysis phase, the algorithm traverses every rule, starting with those whose rule heads that match the query predicate, to build a recursive tree structure called the *Rule/Goal* graph. This graph consists of *rule* and *goal* vertices. A goal vertex consists of a predicate with an adornment indicating which of the attributes in the predicate are bound and which free. A rule vertex represents the bound/free state of all seen variables within a rule body, up to a particular position in its left-to-right execution. A rule with k body predicates will result in exactly k rule vertices corresponding to its positions $[0, ..., k-1]$.

Figure 9 illustrates the full graph for our example. To build the graph, the algorithm starts with a goal predicate (at first, the query predicate—`path` in our example), and creates a goal vertex in the graph with the appropriate adornment ($bbff$ for `path` since the query binds its first two variables). For every rule with that goal predicate as its head, the algorithm traverses the rule body from left to right, creating a child rule vertex for every 0-th position variable binding. For rule `r2` in the example, the rule vertex for position 0 ($r_{2,0}$) has variable state $[X, Y|P, C]$, which denotes that variables `X,Y` are bound (to the same values as those "pushed down" from the goal vertex) and `P,C` are free. Given a rule vertex for position i, two children are created: a goal vertex for the next predicate after position i and a rule vertex for the next position in the rule (unless the body's end has been reached). In the running example, the child goal vertex corresponds to the `link` predicate that appears at position 0, with adornment bff since `X` is already bound at this point in the evaluation, but `Z, C1` are not. Similarly, the child rule vertex at position 1 contains the variable signature $[X, Y, Z, C1|P, C]$ since `link` added some bound variables. The process continues until all rule and goal vertices have been constructed and connected; only a single goal vertex can exist with the same predicate and adornment, as is the case for the `path` vertex with adornment $bbff$. It is easy to see how the rest of the rule/goal graph is constructed. After all

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

r1_g3a path(@X, Y, P, C) :-
          magic_path(@X,Y), link(@X,Y,C), P := f_cons(X,Y).

r2_g1a magic_path(@X, Y) :-
          sup_r2_1(@X, Y, Z, C1).

r2_g3a sup_r2_1(@X, Y, Z, C1) :-
          magic_path(@X, Y), link(@X, Z, C1).

r2_g3c path(@X, Y, P, C) :-
          sup_r2_1(@X, Y, Z, C1), path(@Z, Y, P2, C2).
          f_contains(X, P2) == false,
          P := f_cons(X, P2), C := C1 + C2.

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

Figure 10: A magic-sets rewrite of the rules in Figure 8 (materialize statements not shown).

vertices have been generated given the chosen rules, any predicates with a unique adornment in the graph are added to the goals and the rules producing them are recursively traversed.

In the program rewrite phase, Ullman's algorithm traverses the rule/goal graph generating magic predicates for each "goal" vertex that is unique for its (IDB) predicate; in the example, there are multiple vertices with different adornments for `link`, but only one for `path`, so only `path` is chosen. This magic predicate is inserted in the 0-th position of all rules with the corresponding goal predicate as the rule head, with the bound variables of the signature as attributes. In the example, the magic predicate for `path` has the form `magic_path(@X, Y)` since `X, Y` are the bound variables in the signature of the "goal" vertex. Also *supplementary* predicates are similarly created for all encountered "rule" vertices during the graph traversal and inserted within the corresponding original rule. For example, `sup_r2_1(@X,Y,Z,C1)` is created for "rule" vertex $r_{2,1}$ with the bound variables of the adornments in the vertex, and placed in the original rule `r2` between the `link` and `path` predicates.

Finally, in the filter population phase, the algorithm maintains the magic predicate relation, which was placed within the rewritten program in the previous phases. 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(LOCALHOST, "localhost:10000").`" 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_g1a` [7] adds more magic facts as more `sup_r2_1` tuples are produced.

Our rewrite implementation of this algorithm first traverses every rule from head predicate to body predicates from left to right, constructing the rule/goal graph in the recursive manner of the program analysis, in a single fixpoint. Then the new program rules (and replacement of old rules) for the program rewrite and filter population phases are performed via a traversal of the newly constructed rule/goal graph in a subsequent fixpoint. Finally, initial

[7]Rule names that deal with magic and supplementary predicate maintenance were named according to Ullman's rule groups. For instance, rules named `r*_g3[a-c]` follow rule group 3 and rule `r2_g1a` follows rule group 1.

```
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, PredName, a_count<*>) :-
          idbPredicate(@A, Pid, PredName),
          adornment(@A, Pid, Rid, Pos, PredName, Sig).

mg2 magicPred(@A, Pid, GoalName, Sig) :-
          goalCount(@A, Pid, GoalName, Count),
          adornment(@A, Pid, _, _, GoalName, 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, PredName, Sig) :-
          supNext(@A, Pid, Rid, Pos, PrevSupSchema),
          idbPredicate(@A, Pid, PredName),
          rule(@A, Rid, Pid, _, _, _, _, _),
          predicate(@A, _, Rid, _, PredName, _, _,
                    Schema, Pos, _, _),
          Sig := f_adornment(PrevSupSchema, Schema).
```

Figure 11: Rule/Goal graph traversal rules.

magic facts are created by direct translation from the query. Other details we elide here for brevity involve detecting eligibility of a predicate for a magic-sets rewrite (whether or not it has a unique adornment in the rule/goal graph), state cleanup, etc.

To give a flavor of the Overlog implementation of magic-sets, Figure 11 shows six rules that build the state necessary in the magic-sets rewrite by traversing the rule/goal graph. The `adornment` predicate contains the predicate name ($PredName$) 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 [8]. 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

[8]Two rules (not shown) move the `supNext` position forward if the given rule position does not identify a predicate.

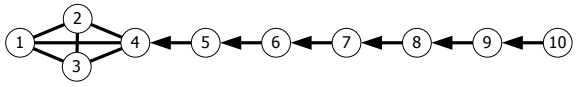

Figure 12: Experimental topology.

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.

4.2.2 Magic-sets in the Network

With the details of the magic-sets algorithm behind us, what is intuitively happening to the shortest-path snippet in Figure 10 is that variable bindings in the query are recursively translated into filtering magic and supplementary predicates. Since the query is only looking for paths to destination "localhost:10000", at first the magic fact restricts single-hop paths created from links in rule `r1`) to only those with that same destination (in the rewritten rule `r1_g3a`). Similarly, in what used to be rule `r2`, `link` tuples are filtered according to the magic predicate (in rule `r2_g3a`), before being joined with existing `path` tuples to complete the old rule `r2`. The reason rule `r2` was split into the two rules `r2_g3a` and `r2_g3c` is because the supplementary result `sup_r2_1` is useful towards adding extra bindings as magic tuples (in rule `r2_g1a`); 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 recursive, then such recursive definitions of magic facts would not appear in the rewritten program.

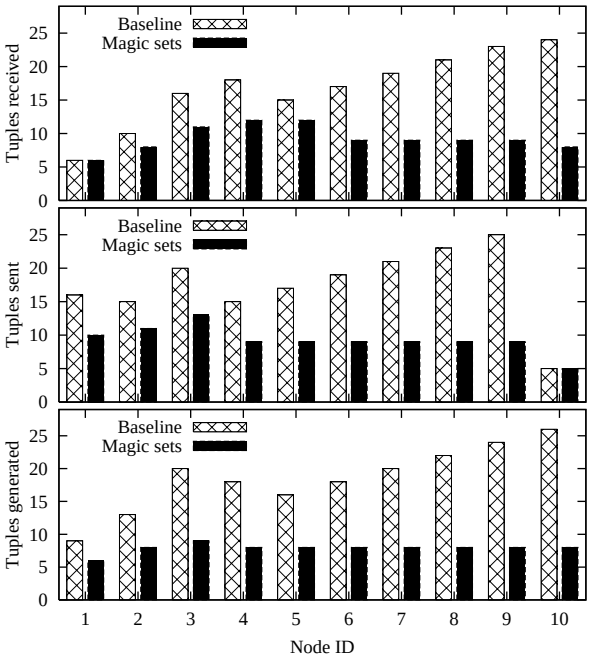

Figure 13: For each node (node ID on x axis), number of tuples received (top), sent (middle), and locally generated (bottom) on the y axis.

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 Section 4.3 since they are distributed). The two programs are executed in the simple link topology of Figure 12. 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 13(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 1 are pruned early on and never transmitted all the way to the far end. Similarly, Figure 13(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). 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, it transmitted tuple overhead is unaffected, since it already sends out no extraneous paths other than its sole path towards node 1. Finally, tuple storage is impacted beneficially by magic sets everywhere (Figure 13(c)), since both `path` tuples received from the network, but also those generated locally for local consumption are pruned away by the rewrite.

4.3 Localization

Finally, we briefly describe the localization compiler stage, which turns a rule with multiple location specifiers in its body to many rules, each of which has a single location specifier in its body; this essentially turns a distributed join into a set of local joins with partial result transmissions among the rules involved [18]. This rewrite is part of the P2 system, but implemented in C++ and woven into the monolithic compiler.

In Evita Raced, localization stage traverses distributed rules in left-to-right order starting with first body predicates (rules with local-only body predicates are selected out early in the stage). The location attribute of the current predicate in this traversal is noted at each iteration. A *breakpoint* is generated if the traversal reaches a predicate with location attribute that differs from the previous. The breakpoint triggers a *split* rule at the given position, which creates a new *glue* predicate IR_p, and two new rules defined as follows.

1. IR_p :- (predicates to the left, excluding the breakpoint).
2. (original rule head predicate) :- IR_p, (predicates to the right, including the breakpoint).

The location attribute in the IR_p predicate is taken from the predicate at the breakpoint position. The other attributes in the IR_p predicate are taken from the predicates to the left of (and not including) the breakpoint, which represents the schema of the intermediate result prior to the breakpoint position predicate. The algorithm then removes the original rule, and moves recursively on the second rule, which is considered the original rule in the above discussion. The recursion terminates at the rightmost predicate position.

The localization stage is not an optimization per se, but rather a program rewrite necessary to make distributed rules executable.

The Overlog program for localization consists of only 28 rules. The original P2 code that performed this task in P2 consisted of approximately 400 lines of C++.

5. RELATED WORK

The pioneering work on extensible query optimizer architectures was done in the EXODUS [9] and Starburst [17,22] 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 [12] and Cascades [10]) 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 fixed in procedural code. By contrast, Evita Raced does not embed a search algorithm, instead leaving that open to specification as needed. The dynamic programming search we implemented is a natural fit to a Datalog-based rule language, and it is an interesting future work challenge to consider whether the Cascades-style "top-down" optimization is easy to achieve in Overlog; the fact that magic-sets rewriting blurs the difference between top-down and bottom-up logic evaluation may be germane here [23].

Another interesting extensible query optimizer is Opt++ [13], 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.

The cyclic dataflow used for stage scheduling in Evita Races resembles the continuous query engine of TelegraphCQ, with our StageScheduler and Demux elements working together to behave somewhat like the TelegraphCQ *eddy* operator [6]. 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.

Our description of Overlog was based on our understanding of the current state of the P2 codebase. Loo et al. [18] describe a language for P2 they call Network Datalog (NDLog), that is roughly a simple sub-language of the current state of Overlog. NDLog is a subset of Datalog with aggregation, and hence does not provide delete rules or updates. Unlike Overlog, NDLog offers well-defined global program semantics across the network, but it does so by not offering delete or update rules, which are used in many Overlog programs.

6. EXPERIENCES

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 has been 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.

P2's notion of consecutive Datalog-style fixpoints, especially in networked environments, still has 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 is akin to a single iteration through a recursive query evaluation strategy like semi-naive 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 [21]. P2's Datalog-style fixpoints are implemented as sequences of such single-event iterations, in a manner that appears to have been an afterthought. As a result, the system's design shares both event-driven and logic-style flavors, with some remaining unresolved conflicts, and no explicit language constructs to bridge between the two.

One example is the notion of `delete` rules, the semantics of which are unclear. How is one to handle delete rules triggered by the *deletion* of a base tuple? The system certainly does not support – semantically or operationally – the "undeleting" of tuples that were originally deleted due to a base fact that is no longer in the database. Similarly, the semantics for multiple updates to the same tuple within the same fixpoint are undefined and a local tie breaking rule is chosen to decide on a consistent ordering among same-fixpoint updates to the same relation. Compiler stages that do static analysis might catch such dangerous rules and alert the user.

Second, as in most prototypes, the programmer interface is not polished. Debugging is difficult, especially since the logic language makes it tough to understand which value corresponds 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 has a long way to go. The language still offers no modularity, making it tough to isolate and reuse logically distinct components. It does has a rudimentary concrete type system, but has poor support for structured types like matrices and lists. Overlog still "cuts corners" on the proper set-orientation of Datalog; since program stratification is only preliminary in the system prototype, dealing with streaming aggregates in the face of EDB updates required us to resort to imperative tricks like timers and polling to determine that aggregates were 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 that ease this interaction may relieve 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 be-

yond localization. As Overlog and P2 mature, the use of a meta-compilation approach should get even easier. And based on our initial experience extending Overlog with security properties in a manner similar to [3], we believe that our Evita Raced infrastructure could accelerate the ability of the P2 group to pursue modifications to Overlog itself.

7. CONCLUSION AND FUTURE WORK

The Evita Raced meta-compilation framework allows Overlog program transformations to be written in Overlog and executed in 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 core query processing 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 two of the most important optimizer frameworks – System R and Magic Sets – in only a few dozen rules each.

Going forward, we hope to exploit the use of a declarative language for benefits beyond code compactness. The tabularization of the optimizer state is particularly suggestive. It can be used to enable optimizer debugging via interactive queries or standing alerts (watchpoints) on the optimizer tables. We are also considering the possibility of implementing adaptive query processing schemes by manipulating the optimizer state, especially given the similarity between our StageScheduler and the eddy operator [6]. Evita Raced is fully operational, and on a more pragmatic front we plan to write many additional rewrites in Overlog, including proper program stratification, integrity constraint implementations, and multi-query optimizations.

8. REFERENCES

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

[4] 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.

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

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[7] 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.

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

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

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

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

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

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

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

[15] 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.

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

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

[18] 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.

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

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

[21] Y.-E. Lu. *Distributed Proximity Query Processing*. PhD thesis, University of Cambridge, Cambridge, UK, 2007. Under review.

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

[23] R. Ramakrishnan and S. Sudarshan. Top Down vs. Bottom Up Revisited. In *International Logic Programming Symposium*, 1991.

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

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

[26] R. Soule, R. Grimm, and P. Maniatis. Auto-Parallelization for Declarative Network Monitoring. In *SOSP Poster Session*, 2007.

[27] J. D. Ullman. Lecture Notes on the Magic-Sets Algorithm. `http://infolab.stanford.edu/~ullman/cs345notes/slides01-16.pdf`. Fetched on 11/15/2007.

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