Declarative Optimizations for Distributed Query Processing Engines

Tyson Condie
U.C. Berkeley

Yu-En Lu
University of Cambridge

Joseph M. Hellerstein
U.C. Berkeley

Petros Maniatis
Intel Research, Berkeley

ABSTRACT

Distributed query processing is a primary bottleneck in many applications that interpret data distributed over a large set of machines. The distributed query engine component is often part of the application code base, and finely tuned to handle a very specific workload and network topology. Performance suffers when assumptions about the data or system state deviate from the norm. Moreover, the distributed query engine logic complicates the application design by incorporating code that queries massive amounts of data spread across a large number of machines.

In this paper we describe the P2 declarative query optimizer meta-compilation framework. The optimization framework is contained within the P2 query processing engine, and exports an interface for submitting rewrite rules over the logical query plan. A rewrite rule takes the form of a query written in the same declarative language (Overlog) used to submit client queries. The logical query plan of a query is represented by a set of relations, which the rewrite rules query and update. In this paper, we show that many traditional database optimizations, like the System R optimization algorithm and the Magic-sets rewrite, can be easily expressed as Overlog queries. We also show that statistics gathering, for cost based optimization rules, can also be written as Overlog queries over the data and system state, leading to a more adaptive query plan generator. Expressing the optimization algorithms as Overlog queries leads to a more concise representation of the algorithm and a dramatic reduction in the overall coding effort.

1. INTRODUCTION

2. RELATED WORK

System R optimizer...
Volcano optimizer....
Opt++ optimizer....

Declarative languages have been used for data processing since the early days of relational databases, leading to the widely-used SQL query language. In the 1980's, significant research effort in the deductive database literature was spent on extending declarative languages with powerful features like recursion, and various optimization techniques were invented for those languages as well. However, the practical use of recursive queries has been quite limited in the field until a recent revival of applicability in a number of research areas [**?**].

One of the most promising new application areas for such recursive queries has been declarative networking, in which declarative programs are used to specify and implement networks and distributed systems. This work has gained the attention of the networking [**?**, **?**], distributed systems [**?**, **?**], and database [**?**] research communities. The P2 group began this work two years ago by looking at *declarative routing* [**?**], demonstrating that recursive queries can be used to express a variety of well-known wired and wireless routing protocols in a compact and clean fashion, typically in a handful of lines of program code. This compactness arises in part from the centrality of graphs and graph algorithms in networking; recursive queries were designed precisely for queries about graph properties.

Encouraged by the ability to compactly specify protocols in a logic language gave rise to a system called **P2** (`http://p2.cs.berkeley.edu`) that uses this language to ease the construction of the application-level overlay networking logic often implemented in distributed systems on the Internet [**?**]. P2 takes specifications in a declarative query language, and compiles them into dataflow programs that resemble a mixture of traditional relational query plans and traditional network routers. P2 then installs those dataflow programs onto a set of machines in the network, resulting in distributed execution of the desired protocol.

We developed an implementation of the Chord [**?**] content-based routing network (also known as a "Distributed Hash Table" or DHT) specified in 47 Datalog-like rules, versus *thousands* of lines of C++ for the original version from MIT; the declarative implementation was competitive with the original in terms of performance and robustness [**?**]. P2 has since been used by multiple researchers both within and outside the group for a wide variety of additional tasks including distributed system debugging [**?**], distributed consensus protocols [**?**] and parallel data-intensive computations that promise to significantly generalize data-parallel systems like Google's MapReduce [**?**].

3. MOTIVATION

- Extending our work in Declarative Networks toward a Declarative Optimizer.. - Carry forward the benefits shown in Declarative Networks to an optimizer framework.

4. P2 ARCHITECTURE

In this section we describe the Overlog language and the compilation of an Overlog program to an execution plan. Our presentation of Overlog will be given in the context of its original purpose;

VLDB '07, September 23-28, 2007, Vienna, Austria.

defining network protocols.

4.1 P2 System Components

The design of P2 was inspired by prior work in both databases and networking. It is based in large part upon a side-by-side comparison between the PIER peer-to-peer query engine [**?**] and the Click router [**?**]. Like PIER, P2 can manage structured data tuples flowing through a broad range of query processing elements, which may accumulate significant state and perform substantial asynchronous processing. Like Click, P2 stresses high-performance transfers of data units, as well as dataflow elements with both "push" and "pull" modalities.

At a coarse grain, P2 in its current state consists of (1) an Overlog parser, (2) an Planner that translates Overlog to a runtime dataflow plan, and (3) a runtime plan executor. The life of a query is simple: the query is parsed into an internal representation, the planner constructs a corresponding dataflow graph of elements, and the graph is executed by the runtime until it is canceled. We proceed to overview the components bottom-up; more details are given in the P2 SOSP paper [**?**].

Processing in P2 is handled with a dataflow model inspired by Click and PIER. As in Click, nodes in a P2 dataflow graph can be chosen from a set of C++ objects called *elements*. In database systems these are often called *operators*, since they derive from logical operators in the relational algebra. Elements have some number of input and output *ports*. An arc in the dataflow graph is represented by a binding between an output port on one element and an input port on another. Tuples arrive at the element on input ports, and elements emit tuples from their output ports. Handoff of a tuple between two elements takes one of two forms, *push* or *pull*, determined when the elements are configured into a dataflow graph.

P2 provides a number of built in dataflow elements that allow it to implement networking and query processing logic. This includes elements for the streaming relational query operators found in most database systems, e.g., selection, projection, join, and aggregation. It also includes networking elements responsible for socket handling, packet scheduling, congestion control, reliable transmission, data serialization, and dispatch. P2 has elements to store incoming tuples in tables, iteratively emit tuples in a table matching a filter expression, and *listener* elements that are notified whenever a tuple is added or deleted from a table. Finally, like Click, P2 includes a collection of general-purpose "glue" elements, such as a queue, a multiplexer, a round-robin scheduler, etc.

Storage in P2 is currently via a main-memory relational Table implementation, named using unique IDs that can be shared between different queries and/or dataflow elements. In-memory indices (implemented using standard balanced binary trees) can be attached to attributes of tables to enable quick equality lookups. The current in-memory implementation serves the system requirements for implementing network overlays and streaming query applications, all of which tend to expire tuples from memory rather than accumulating them indefinitely. P2's event-driven, run-to-completion model obviates the need for locking or transaction support, and relatively simple indices suffice to meet performance requirements. We plan additional table implementations that use stable storage for persistent data storage; that engineering task is relatively straightforward, but not within the scope of this paper.

4.2 Physical Plan Generation

Our current Overlog parser is fairly conventional, implemented using `flex` and `bison`. It converts Overlog text into a canonical form and produces data structures for rules and table definitions. The current P2 planner takes these data structures and translates each rule into a *dataflow strand*: a directed graph of dataflow elements. This is currently done in a manual (and fairly ad hoc) way as follows. First, all the required tables and indices are created. We create an index for every table's primary key, and secondary indices for table keys participating in equijoins. Then, each rule is passed to a *localization rewrite* component that rewrites the rule into one or more new rules, each of which can be evaluated on a single machine, though its output may be sent remotely; it is shown in [**?**] that this localization rewrite is always possible for well-formed Overlog. The resulting localized rules are rewritten using the Pipelined Semi-Naive (PSN) technique described in that paper. PSN converts each Overlog rule into a set of Event-Condition-Action rules, each of which listens for changes on a single table, propagates those changes through a dataflow strand of conditions (which often includes "equijoin" table lookups), and generates tuple-derivation "actions" that are translated into local table updates and/or network messages depending on the appropriate destination of the tuple. In the end, the P2 dataflow runtime need only be able to handle these ECA strands.

In addition to creating the relational operations described above, the planner also constructs the other areas of the dataflow graph: the networking stack, including multiplexing and demultiplexing tuples, marshaling, congestion control, etc. As with Click, it also inserts explicit queue elements where there is a push/pull mismatch between two elements that need to be connected.

4.3 P2 Operational Semantics

Unlike many Datalog variants which operates on a static database, an Overlog program actively update its base tables due to *external* events. In P2, events are external if they are due to table updates, network messages, or periodic timer events. Internal events are all the local site consequences due to external events. We provide an operational semantics that guarantees atomic fixpoint evaluation of any external event. We find this semantics eliminates the need for locking or transactional support for most single site programs and gives us a foundation on which global fixpoint evaluation could be built. For example, a log-entry tuple could be inserted to local log table to mark the state at which a on-going two-phase commit protocol is. Since network messages are external, a commit vote cannot be sent until all pending state changes are applied.

In this section, we concentrate our discussion on the local site, single event fixpoint evaluation currently offered by P2 engine.

4.3.1 Overlog in a Nutshell

In order to understand what a local event looks like in the P2 world we first begin with a rough overview of the its declarative language. The Overlog language was developed in order to specify network protocols following largely the Datalog language. An Overlog program consists of a set of declarative *rules* each of which is of the form Event-Condition-Action. Specifically, the right-hand-side of the rule contains a set of conjunctive predicates over relations. This set of predicates contains exactly one event which is implicitly either *external* or *internal*. The left-hand side represents the deduction from the right-hand side which we shall refer to as action. An action can be table updates, generation of network messages or local event tuples.

Overlog extends the standard Datalog in two ways. The first is the addition of a location specifier in all tuples. The location specifier indicates the location of the tuple. This enable us to incorporate remote relations and generate new tuples for remote evaluation in a rule. The second addition is the inclusion of an event predicate in each rule. The event predicate triggers the execution of the rule as well as specifies the attribute variable of the local node. erm, I am

not so sure about event being the new addition. comments?

```
E1:  action(@S, A, B, C)
         :- event(@S,A), condition1(@S, A, B),
         condition2(@S, B, C).
E2:  action(@C, A, B)
         :- event(@S, A),condition(@S, A, B, C).
```

Figure 1: ***Overlog example.***

Figure 1 provides three example Overlog rules. The left-hand-side of each rule is given the name `action`, which can represent a update to the database or the occurrence of an event. Both the database updates and event actions may occur either locally or remotely depending on the location attribute. Confirm: Are we allowing `delete action(@Y):- event(@S,Y)`!!!??? That would require a distributed fixpoint! The right-hand-side contains an event followed by a set of predicates. Each predicate is represented as a tuple of values referenced by the predicate attributes. For example, in rule *E1* the `event(@S, A)` predicate will be represented by a tuple named `event` with two values. The first is the location of the local node and the second is some atomic value referenced by the variable `'A'`. Upon receipt of a such a tuple, the query engine will perform a join with the tables referenced by `condition1` and `condition2`. The join attributes are resolved by name. That is, the tuple $< event, @S, A >$ joins with $< condition1, @S, A, B >$ along attributes $@S$ and A. Results from that join will be joined with tuples in the table referenced by `condition2`. The final join results will be applied to the `action` portion of the rule. If the rule head (`action`) holds a different location specifier than the event (right-hand-side) then the resulting tuples will be sent to a remote site and applied accordingly (as an event or database side effect). Rule *E2* in the aforementioned example would result in the tuples being sent across the network to their respective location specified by `'C'`.

4.3.2 Atomic Local Fixpoint Evaluation

```
E3:  event2(@S, A, B, C)
         :- event1(@S,A),
         condition1(@S, A, B), condition2(@S, B, C).
E4:  action(@C, A, B)
         :- event2(@S, A),condition1(@S, A, B, C).
```

Figure 2: ***Internal event example.***

Executing an event to fixed point requires a further distinction between two types of events. An *external* event occurs due to network messages, change in the database tables, and timer events. For the purposes of this survey, we will only focus on events triggered by the arrival of some tuple over the network. An *internal* event is triggered by the *action* portion of a rule, specifically when the action does not result in a table side effect or a remote send operation.

For example, Figure 2 shows two example rules named *E3* and *E4*. The action portion of rule *E3* specifies the assertion of an event named *event2*. That is, an event of type *event1* will trigger rule *E3*, and any resulting tuples from that rule will generate events of type *event2*, each of which will trigger *E4 before* any other external event is considered in the evaluation. This semantics could be summarised as the definition below:

Definition (Atomic Local Fixpoint) An local fixpoint is the program state such that no further deductions are possible without introducing new external events. It is atomic if for each iteration, the following invariants hold:

- the triggering external event and all induced events see the same system state for the fixpoint computation.
- all actions deduced are applied atomically at the end of the iteration

Given the above definition, the fixed point evaluation of `event1` in Figure 2 is the execution of rule *E3* followed by the execution of rule *E4* for each tuple of type *event2* from rule *E3*. If the action portion of rule *E4* is a remote send or a table side effect, it will be applied only when no further local events could be deduced. Otherwise, the internal events resulting from the action portion of rule *E4* will be evaluated immediately. Safety checks have been put in place in order to ensure a fixed point evaluation strategy applied to a given set of rules and events does not cause deadlock. XXX: Eric is still working on the proofs for below. TBC. The fixed point evaluation of an event will always terminate so long as the right-hand-side contains no negated logic. Termination can still be guaranteed when the right-hand-side contains negation so long as the rules can be stratified [**?**].

The execution of an external event and all consequential internal events can cause table side effects and network messages to occur from the actions from some rules triggered during the fixed point evaluation. The P2 system treats ensures that they are not visible until the end of the fixed point after which new external events might arise due to, for instance, table updates. The system internals will buffer all such actions in FIFO order until all internal events have been exhausted. Once that occurs, the buffered (external) actions will be applied, potentially resulting in database side effects and remote send operations. When all actions have been flushed, the P2 system scheduler will begin another fixed point evaluation of the next external event. This behavior is reminiscent of sieves, and guarantees that local Overlog execution is deterministic under the same order of external events. (Arbitrary Overlog computations are non-deterministic due to external events such network delays and tuple reordering.)

4.4 Declarative Networking: An Example

```
D1:  path(Src,Root,Root,Cost) :-
         intree(Src,Root), link(Src,Root,Cost).
D2:  path(Src,Root,Nexthop,Cost) :-
         intree(Src,Root),link(Src,NextHop,C1),
         path(NextHop,Root,Hop2,C2), Cost = C1 + C2.
D3:  minCost(Src,Dst,min<Cost>) :-
         path(Src,Dst,NextHop,Cost).
D4:  parent(Src,Root,NextHop,Cost) :-
         minCost(Src,Root,Cost),
         path(Src,Root,NextHop,Cost).
Query: parent(Src,Root,NextHop,Cost).
```

Figure 3: ***Tree routing query in traditional Datalog.***

To illustrate declarative networking, we begin with a simple example that specifies shortest-path routing trees from a set of nodes to a set of one or more tree roots. Tree construction protocols are frequently used for aggregation queries, among other tasks. In Figure 4.4 we present this protocol in the traditional recursive query language *Datalog* enhanced with aggregation functions [**?**]. Datalog programs consist of a set of declarative *rules*, terminated by periods. The right-hand-side of the rule represents a conjunctive predicate over relations in a database, and the left-hand side represents the deduction from that predicate. For example, rule **D1**

```
ND1:  path(@Src,Root,Root,Cost) :-
        intree(@Src, Root),
        #link (@Src,Root,Cost).
ND2:  path(@Src,Root,Nexthop,Cost) :-
        intree(@Src, Root),
        #link (@Src,NextHop,C1),
        path(@NextHop,Root,Hop2,C2),
        Cost = C1 + C2.
ND3:  minCost(@Src,Dst,min<Cost>) :-
        path(@Src,Dst,NextHop,Cost).
ND4:  parent(@Src,Root,NextHop,Cost) :-
        minCost(@Src,Root,Cost),
        path(@Src,Root,NextHop,Cost).
Query: parent(@Src,Root,NextHop,Cost).
```

Figure 4: ***Tree routing in Overlog*.**

can be read as "**if** there is a tuple `(Src, Root)` in the `intree` relation, **and** there is a tuple `(Src, Root, Cost)` in the `link` relation (where the `Src` and `Root` variables of the two tuples match), **then** there is a tuple `(Src, Root, Root, Cost)` in the `path` relation." This rule identifies paths to the root of each tree from that root's children; those paths have the cost of the corresponding one-hop link from child to root, and the paths' "next hop" for routing (the third field of the `path` relation) is, naturally, the root itself. This is the base case of a recursive path-finding specification. The recursive case is captured in rule **D2**: **if** a source node has a link to another "NextHop" node, **and** that node in turn has a path to the tree's root, **then** the `src` node has a path to the root *via* that "NextHop" node, with the appropriate cost.

These two rules are sufficient to find all possible paths from sources to tree roots. The next two rules prune this set: **D3** uses the aggregation function `min` to identify the cost of the shortest (least-cost) path from each source to each root, and **D4** sets the `parent` of each source in each tree to be the `NextHop` in the shortest path. Finally, the query specifies that all such `parents` should be returned as output.

The short Datalog program of Figure 4.4 is sufficient to specify the establishment of shortest paths to tree roots in a graph of multiple trees, and a Datalog engine can naturally translate each tree's path-finding rules into an efficient shortest-paths algorithm like Dijkstra's. However, this is still not a network protocol: Datalog assumes that the relevant data, and the query processing computation, are centralized on a single computer. Figure 4 shows a variant of the program expressed in our *Overlog* language, which specifies a workable, distributed network protocol based on that logic. There are only two modifications introduced in Figure 4:

1. Each relation has one field prepended with the "`@`" symbol; this field is called the *location specifier* of the relation. The location specifier specifies data distribution: each tuple is to be stored at the address in its location specifier field. For example, the `path` relation is partitioned by the first (source) field; each partition corresponds to the networking notion of a local routing table.
2. A special class of *link* relations have names beginning with "`#`", and contain two fields of type `address` representing source and destination nodes in the network. These predicates capture edges in the underlying network graph. The link predicate in Figure 4 is called `#link` , and corresponds to the networking concept of a local neighbor table.

Loo et al. [**?**] show how location specifiers and link relations enable an Overlog compiler to generate a distributed protocol that is guaranteed to be executable over the underlying network topology captured by the link relation.

Table 1: System catalog table definitions that are relevant to the compiler.

Table Name	Foreign Key
Program	none
Table	Program
Attribute	Table
Rule	Program
Table Predicates	Rule
Selection Predicates	Rule
Assignment	Rule
Rewrite	Program

5. DECLARATIVE OPTIMIZATION

This section describes the extensions made to the P2 architecture that allow programers to specify optimization rewrites written in Overlog. In order to accommodate optimizations expressed in Overlog we needed to express the structure of a logical query plan in relational form. The transformations performed by the rewrites take the form of queries and updates over the tables that make up the logical query plan. Once all relevant rewrites have executed, the physical plan is generated by again querying the tables that describe the logical query plan. The following sections describe the compilation process. We conclude this section with some example optimization rewrites.

5.1 Meta-compilation Interface

The optimizer interface is identical to the regular query interface. A programmer of a rewrite submits an Overlog program to the compiler, which will be installed as a compilation stage. In order to determine when the rewrite can be applied to a program, the rewrite programmer must specify a list of stage dependencies. This dependency list will be entered into the Rewrite table as one row for each dependency. The Rewrite table represents a lattice of rewrite orderings, and is queried by the optimizer when determining the next rewrite to apply to a Program. Once a rewrite has been installed, it will be able to participate in the optimization process in the order prescribed by the lattice. Since the interface for installing a rewrite is the same as a regular Overlog program, it will receive the benefits of any rewrites that were installed before it.

5.1.1 System Catalog

An optimization is a transformation of a logical query plan into another *equivalent* logical query plan. This process is traditionally performed on some internal tree-like data structure within a procedural programming language. Instead of a procedural language, our optimizations are written in Overlog as queries and updates over a set of tables representing the logical query plan. Table 1 lists the table and foreign key references presented to the programmer of an optimization.

5.2 Meta-compilation Architecture

The compiler is comprised of a set of compilation stages, each of which operate on the logical query plan in a manner defined by the user. A compilation stage can be written in either C++ or Overlog. The stages written in C++ make up the initial components of the compiler and must be installed at system bootstrap to provide minimal Overlog program compilation support. Further compilation stages written in Overlog extend the compiler functionality with optimizations, error and safety checks, and various forms of program analysis.

A compilation stage is scheduled by the *Compilation Scheduler*,

which resides in the dataflow as an element and listens for events of type *programEvent*. This event contains attributes relevant to the program under compilation. One of these attributes indicates the stage that is currently operating on the program. This attribute is updated by the scheduler, according to the stage lattice described in Section ??, and reasserted to dataflow, which will route the event to the respective compilation stage. A compilation stage that receives such an event will perform its task on the program referenced by the event attributes. Upon completion of its task, the compilation stage will assert a new *programEvent* to the dataflow, which will be routed back to the scheduler, and the process proceeds with the following compilation stage.

In the beginning, the first compilation stage chosen by the scheduler is the parser. It takes the text description of the Overlog program contained in the *programEvent* and inserts tuples in the tables that define the logical query plan. The final two compilation stages are the physical plan generator (planner) and the dataflow compiler. The planner generates a dataflow text description of the logical query plan and adds this description to the *programEvent*. The dataflow compiler compiles the dataflow description by instantiating and installing the necessary dataflow elements that make up the physical plan. The remainder of this section describes the components that comprise the initial set of compiler stages, all of which must be installed at system bootstrap due to physical plan generation dependencies.

5.2.1 Parser

The Parser is the first stage in the compilation pipeline. It takes a text description of an Overlog program as input and performs a traditional parse using flex and bison. If the program is syntactically correct, the Parser inserts a set of rows into the tables that represent the logical query plan. A parse error will result in an error event that will be returned to the sender of the program.

5.2.2 ECA

The ECA stage determines the parts of a rule that represent the event, condition, and action. The stage ensures that the rule contains a single event, zero or more conditions, and a single action. Any rule that does not contain an event (such as materialized views) require an addition rewrite be installed that transforms such rules into equivalent rules that each contain an event. Such a rewrite is presented in Section **??**. This stage is currently written in C++ because of dependencies written in to the physical plan generator.

5.2.3 Physical Plan Generator

Physical plan generation occurs once all stages have been applied to the logical query plan. The planner generates a physical plan by querying the set of tables describing the logical query plan. The physical plan is described by a P2 Dataflow Language (P2DL) program, which is compiled and installed in the final stage.

5.2.4 P2 Dataflow Language Compiler

An physical query plan is described by a dataflow language that define the physical operators (elements) that implement a given Overlog program. The language specifies the initialization parameters to the elements and how elements are to be connected to form the physical query plan. We have developed a compiler that accepts a text value description of the dataflow graph. From the text description, the compiler will instantiate and install the physical query plan in to a running P2 node.

5.3 Overlog Rewrite Compilation Stages

In this section we describe some of the rewrites that we have written in Overlog and installed as compilation stages in the compiler. A rewrite stage requires no special system support for installing its physical dataflow plan. It does require that the rewrite programmer follow a simple coding convention that indicates when the rewrite should begin and end. As mentioned in Section ??, a rewrite is triggered by a *programEvent* assertion, which contains attributes that describe the Overlog program currently under compilation. Once a rewrite has completed its task it must generate a new *programEvent*, which will be routed to the compilation scheduler for the scheduling of the next rewrite.

```
L1:  foo(@S, A, B, C) :-
         event(@S, A, B), link(@S, T, U),
         bar(@T, A, B), baz(@U, B, C).

L2a:  intermediateS(@T, S, A, B, U) :-
         event(@S, A, B), link(@S, T, U).
L2b:  intermediateT(@U, S, A, B, U) :-
         intermediateS(@T, S, A, U), bar(@T, A, B).
L2c:  foo(@S, A, B, C) :-
         intermediateT(@U, S, A, B, U), baz(@U, B, C).

L3a:  intermediateS(@U, S, A, B, T) :-
         event(@S, A, B), link(@S, T, U).
L3b:  intermediateU(@T, S, A, B, C, U) :-
         intermediateS(@U, S, A, B, T), baz(@U, B, C).
L3c:  foo(@S, A, B, C) :-
         intermediateU(@T, S, A, B, C, U), bar(@T, A, B).
```

Figure 5: ***Localization rewrite example*.**

A rewrite contains a set of rules that, among other things, query and update the system tables relevant to the logical query plan of the program currently being compiled. One of attributes contained in the *programEvent* is the program identifier *PID*, which is the primary key to the *Program* system table. Using this key, the rewrite may access the row contained in the em Program table for the current program. To obtain the set of rules in the program, the rewrite may join the *Program* table with the *Rule* table along the *PID* key. The answer to such a query will be all the rule tuples of the respective program. A rule tuple may be joined with the *Select* table to obtain the rule's selection predicates, the *Functor* table for any table predicates the rule contains, or the *Assign* table for assignments that the rule may contain.

5.3.1 Localization

The purpose of a localization rewrite is to convert a distributed join rule into some number of local join rules, along with a set of rules that transfer intermediate results. Clearly, the programmer that posed the original query to the system could perform this task. However, it is often easier to reason about distributed join by posing it as a single rule containing the sequence of join predicates that differ in the location attribute. Breaking the distributed join into a sequence of intermediate results forces to programmer to worry about coordinating a set of nodes to combine the data results of a join in a meaningful manner. Moreover, without a sense of the statistics relevant to the optimizer, the programmer is likely to generate a suboptimal physical plan. This is an example of data independence, which is one of the primary benefits afforded by a declarative language and optimizer, and we believe it will carry over well to many distributed optimization algorithms.

Consider the example rule labeled *L1* in Figure 5. This rule is executed when the *event* tuple is asserted on the site referenced by the $@S$ location attribute. The event tuple is joined with the *link* at the local site as well as the *bar* and *baz* tables at remote sites speci-

fied by the @T and @U location attributes respectively. In order to execute this join the distributed rule will need to be broken up in to a set of rules that perform local joins and ship intermediate values to successive join sites referenced by the location attribute. This can be done in two ways. The first possibility is outlined in rules *L2[a-c]*, which essentially chooses to send all intermediate values from the join of the *event* tuple with the *link* table to the site referenced by attribute @T. At that remote site, the intermediate values are joined with the *bar* table and sent to site U referenced by the @U attribute. The site U receives the intermediate values sent by site T and joins those values with its local *baz* table before shipping the final result back to site S in the form of *foo* tuples. A second possible is outlined in rules *L3[a-c]*, where the only difference is that we first join with tables at site U followed by site T. Choosing between these two plans should be left to the optimizer, and not the programmer.

5.3.2 Materialized View Rewrite

A materialized view rule is a rule that contains only base table predicates and selections in the rule body. Such a rule does not exist in the Overlog model due to the lack of an event in the rule body. The purpose of a materialized view rewrite is to convert such a rule into a set of rules, each of which contain an event, and together are equivalent to the original rule. The process by which this rewrite occurs takes each base table in the body of the original rule and generates a new rule with the following two properties.

1. The event refers to any side effects made to the base table that generated the rule.
2. The condition portion contains all predicates in the original rule, excluding the base table to which the event corresponds.

The Overlog statements that make up this rewrite will generate the new rules according to these two properties.

5.3.3 Isolation Rewrite

5.3.4 Magic-sets

5.3.5 System R

6. EVALUATION

Select operator placement: Scene: A user taking PDAs doing temporature detection, four. Query: When's the last time we had such a temporature across four databases at the same time?

onFire(@Y,X,T) :- onMeasureTemp(@X,TEMP), #link1(@X,A), tTempHistory(@A,TA,T), TA ¿ TEMP, #link2(@X,B), tTempHistory(@B,TB,T), TB ¿ TEMP, #link3(@X,C), tTempHistory(@C,TC,T), TC ¿ TEMP, #link4(@X,D), tTempHistory(@D,TD,T), TD ¿ TEMP.

7. CONCLUSION

8. ACKNOWLEDGMENTS

9. REFERENCES

10. REFERENCES