Entangled Queries: Enabling Declarative Data-Driven Coordination

Nitin Gupta, Lucja Kot, Sudip Roy, Gabriel Bender and Johannes Gehrke
Cornell University
Ithaca, NY 14853, USA
{niting, lucja, sudip, gbender, johannes}@cs.cornell.edu

Christoph Koch
EPFL
CH-1015 Lausanne, Switzerland
christoph.koch@epfl.ch

ABSTRACT

Many data-driven social and Web applications involve collaboration and coordination. The vision of *declarative data-driven coordination* (D3C), proposed in [9], is to support coordination in the spirit of data management: to make it data-centric and to specify it using convenient declarative languages. This paper introduces *entangled queries*, a language that extends SQL by constraints that allow for the coordinated choice of result tuples across queries originating from different users or applications.

It is nontrivial to define a declarative coordination formalism without arriving at the general (NP-complete) Constraint Satisfaction Problem from AI. In this paper, we propose an efficiently enforcible syntactic *safety* condition that we argue is at the sweet spot where interesting declarative power meets applicability in large scale data management systems and applications.

The key computational problem of D3C is to match entangled queries to achieve coordination. We present an efficient matching algorithm which statically analyzes query workloads and merges coordinating entangled queries into compound SQL queries. These can be sent to a standard database system and return only coordinated results. We present the overall architecture of an implemented system that contains our evaluation algorithm; we also evaluate the performance of the matching algorithm experimentally on realistic coordination workloads.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Languages

1. INTRODUCTION

1.1 Declarative data-driven coordination

Collaboration and coordination are increasingly important aspects of the ways people produce, process and consume data. This is true not only for serious tasks such as scientific dataset management, but also at the grass-roots level, as internet users organize and coordinate activities online. In [9], we presented our vision of *declarative data-driven coordination* (D3C) as a high-level design principle for collaborative data management systems. In this paper, we address some of the challenges related to making D3C a reality by introducing a system that supports *entangled queries* – a declarative mechanism for data-driven coordination.

Our paper [9] motivates D3C through a series of real-world coordination scenarios. We briefly revisit these examples here and explain D3C in some detail in order to make more concrete the technical challenges involved in implementing entangled queries.

A common coordination scenario is joint travel planning with friends or family; for instance, several friends might wish to separately reserve seats on the same flight. The desired coordination is based on the attributes of the data itself, such as flight numbers and times, rather than on context information such as the time the booking is made. Thus, the coordination itself is *data-driven*.

There are many other settings in which users wish to coordinate. College students want to enroll in the same courses as their friends, busy professionals want to schedule joint meetings, and wedding guests want to purchase gifts in a way that avoids duplication. Coordination also occurs in massively multiplayer online (MMO) games, where players are often interested in developing joint strategies with other players to achieve common objectives. Again, the coordination is data-driven as it relates to in-game goals.

Despite the ubiquity of scenarios such as those described above, coordination is not commonly supported in today's data-driven applications. For example, joint travel planning usually starts with significant out-of-band communication to fix an itinerary. Next, one designated user makes a group booking, or all users try to make bookings simultaneously and hope that enough seats will remain available. Finally, more communication may be necessary to sort out finances. The same is true for the other examples of coordination mentioned above. In MMO games, for instance, joint strategies are currently formed using out-of-band communication, to the detriment of gameplay experience.

The idea behind D3C is to provide a way for users to coordinate within the system and without having to worry about the details of the coordination. Because the coordination is data-driven, the coordination abstraction is designed to sit at the same level as other abstractions that relate to the data. Declarativity – allowing users to express what is to be achieved, rather than how it is to be achieved – has long been an underlying design principle in databases. In a declarative specification of coordination, the users' only responsibility is to state their individual preferences and constraints, and the system takes care of the rest. D3C is thus in contrast with exist-

SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

ing work on data-driven coordination in workflows [4, 13] and Web services [5, 3, 16], which does not clearly separate the coordination specification and mechanism.

To see what coordination looks like in a system that supports D3C, consider an example. Suppose Kramer wants to travel to Paris on the same flight as Jerry. In our system, he can express his request with the following *entangled query*:

```
SELECT ‘Kramer’, fno INTO ANSWER Reservation
WHERE
fno IN (SELECT fno FROM Flights WHERE dest=‘Paris’)
AND (‘Jerry’, fno) IN ANSWER Reservation
CHOOSE 1
```

Jerry also wants to travel with Kramer, but he has an additional constraint: he wants to travel only on flights operated by United. His query is as follows:

```
SELECT ‘Jerry’, fno INTO ANSWER Reservation
WHERE
fno IN (SELECT fno FROM Flights F, Airlines A WHERE
                   F.dest=‘Paris’ and F.fno = A.fno
                   AND A.airline = ‘United’ )
AND (‘Kramer’, fno) IN ANSWER Reservation
CHOOSE 1
```

Section 2 explains the syntax of these queries in detail. For now, it is enough to understand that Reservation is a name for a virtual relation that contains the answers to all the current queries in the system. The SELECT clause specifies Kramer’s own expected answer, or, in other words, his contribution to the answer relation Reservation. This contribution, however, is conditional on two requirements, which are given in the WHERE clause. First, the flight number in question must correspond to a flight to Paris. Second, the answer relation must also contain a tuple with the same flight number but Jerry as the traveler name. Jerry’s query places a near-symmetric constraint on Reservation.

Neither user explicitly specifies which other queries he wishes to coordinate with – e.g. by using an identifier for the coordination partner’s query. Instead, the coordination partner is designated implicitly using the partner’s query result. This is a deliberate choice that allows coordination with potentially unknown partners based purely on desired shared outcomes. In travel planning, of course, it typically *is* known who one’s coordination partners will be. However in other scenarios such as MMO games, coordination partners may be unknown and their identities irrelevant.

When the system receives Kramer and Jerry’s queries, it answers both of them simultaneously in a way that ensures a coordinated flight number choice. In general, there may be many different suitable flights, but Kramer and Jerry only want to make a booking on one of them. The CHOOSE 1 clause present in both queries specifies that only one tuple is to be returned per query. The tuples returned must be such that all constraints are satisfied. If the database is as shown in Figure 1 (a), the system non-deterministically chooses either flight 122 or 123 and returns appropriate answer tuples. Figure 1 (b) shows the mutual constraint satisfaction that takes place in answering for 122. The intent is that Kramer and Jerry should now be able to make a booking on flight 122.

The above queries are of course simplified to illustrate the basic coordination mechanic; in a real travel reservation setting, they would include checks for seat availability and other factors.

1.2 Enabling D3C

Other research communities have long recognized the need for communication among concurrently running processes and have designed solutions to support it. Systems researchers have developed solutions ranging from low-level mechanisms such as message passing, shared memory, locks and semaphores to higher level abstractions such as transactional memory [10]. The programming languages community has given us Concurrent ML [14], Erlang [18], Stackless Python [1], Concurrent Haskell [8] and many other languages that come with concurrency support. These languages enable communication through channels or other mechanisms in a clean and precisely specified way. At a higher level, abstractions such the π-calculus [12] allow formal modeling and reasoning about communication.

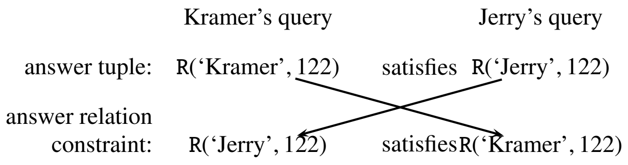
Flights

fno	dest
122	Paris
123	Paris
134	Paris
136	Rome

Airlines

fno	airlines
122	United
123	United
134	Lufthansa
136	Alitalia

(a)



(b)

Figure 1: (a) Flight database (b) Mutual constraint satisfaction

The data management research community has long avoided the coordination problem, probably as a consequence of accepting isolation among transactions as a dogma. However, as pointed out above, data-driven coordination has real uses. The process-centric abstractions mentioned above are not a good fit for data-driven applications [9]; a large class of such applications would be much easier and faster to develop using a data-centric abstraction such as entangled queries. Moreover, a well engineered high-level abstraction like entangled queries creates an opportunity for automatically optimizing coordination on a large scale that is not possible for the lower-level abstractions offered by operating systems.

It is important to emphasize that existing database mechanisms such as nested transactions [11], Sagas [7], or ConTract [15] that weaken isolation in a form or another do not solve the coordination problem, for two reasons. First, they only allow for unidirectional information flow between transactions on the same conceptual layer (of nesting), not the kind of bi- or multidirectional flow required to achieve coordination. Moreover, coordination requires *automated matchmaking* between queries, which is a problem not addressed at all in previous work. It may at first seem that triggers or other active database constructs [19] can address the same problems as D3C, since active databases perform actions based on certain conditions becoming true in the database. However, trigger conditions are preconditions, while the coordination constraints of entangled queries are postconditions on the desired state of the database after the coordination. Again, triggers provide no straightforward way to achieve coordination matchmaking, which is the key problem addressed and solved in this paper.

Once the notion of entangled queries has been formalized, a key technical challenge is to solve the coordination problem. That is, we need an algorithm that finds answers to the entangled queries in a way that satisfies the coordination constraints.

There is, however, a fundamental obstacle. The combination of a declarative query language such as SQL with coordination constraints of the kind illustrated above naturally captures the general Constraint Satisfaction Problem (CSP) of AI [6], which is NP-complete. This source of complexity is included by design: the very idea of D3C calls for a coordination solution to be a *choice* (nondeterministic, if you will) from a query result, *constrained by cross-query conditions*. Declarativity naturally entails a (combinatorial) satisfiability problem.

There are in fact two sources of nondeterminism (disjunction) and thus complexity in the coordination problem. The first is the choice of queries to be grouped together; the second, the choice of data tuples from the query results that are chosen as coordinating solutions. We cannot reasonably hope to eliminate the second type of complexity; this is the same issue that causes select-project-join queries to be NP-complete if one considers the query to be part of the input. On the other hand, one usually considers this acceptable because queries are small. If this second source of NP-completeness had to be eliminated, one could not support declarative queries with coordination constraints in a similar formalism.

A key contribution of this paper is a syntactic condition, *safety*, which ensures that coordination can be performed efficiently in the sense that the first source of complexity is eliminated. Coordination is only NP-hard in the size of the groups of queries or individuals who want to coordinate; in a travel scenario like our example where an arbitrary number of pairs of two people want to coordinate, this size is two. The hardness result is independent of the total number of entangled queries in the system, and also of the size of the data in the database. The latter fact is comfortingly obvious from the fact that the algorithm presented in this paper merges queries to be coordinated statically into standard SQL queries that only produce coordinated solution tuples for the constituent entangled queries; the essential query matching/coordination problem is solved without access to the data.

1.3 Contributions

The contributions of this paper are as follows. First, we formalize entangled queries, a simple yet powerful abstraction for D3C. Entangled queries are expressed in an extension of SQL, allowing the coordination constraints and the data involved in the coordination to be specified at the same level of abstraction. They are inspired by a language example from [9]; however, in this paper we give a full formal treatment of these queries, including a precise syntax and semantics.

Second, we introduce a formal notion of *safety* for queries that are admitted into the system. In keeping with our previous discussion, safe queries are designed to allow efficient evaluation in realistic settings rather than express generic CSP instances.

Third, we present an algorithm for coordination. The algorithm begins by working at the syntactic level to solve the *query matching* problem – identifying the potential coordination partners for each query. Next, each set of matching queries is combined into a larger query that expresses the desired joint outcome. For example, Jerry and Kramer's queries would be combined into a single query asking for a United flight to Paris. Finally, the answers to the combined query are used to generate individual answers.

Fourth, we introduce an end-to-end system that supports entangled queries. Apart from an optimized implementation of the algorithm, we present other components for query management and interaction with the application layer. Our system supports coordination in two modes: set-at-a-time mode (queries arrive in batches) and incremental mode (queries arrive as a stream). We leverage the properties of coordination structures to partition and evaluate query sets independently and in parallel.

Finally, we give experimental results that use our system and demonstrate the scalability of the coordination algorithm. We strive to use workloads that are as realistic as possible; in generating them, we make use of real social network data and extend them to a scale which is realistic for today's internet.

The remainder of this paper is organized as follows. Section 2 introduces the syntax and semantics of entangled queries. Section 3 discusses the kinds of coordination structure that are likely to be present in the most common use cases. Section 4 presents the evaluation algorithm for coordination. Section 5 describes our system implementation and contains experimental results, while Section 6 discusses future work. We mention related work throughout.

2. ENTANGLED QUERIES

In this section, we introduce a SQL-like syntax for entangled queries, propose an intermediate representation for ease of exposition and define the semantics of query answering.

2.1 Syntax

An entangled query is expressed in extended SQL using the following syntax:

```
SELECT select_expr
INTO ANSWER tbl_name [, ANSWER tbl_name] ...
[WHERE where_answer_condition]
CHOOSE 1
```

The `WHERE` clause is a normal condition clause that may refer to both database and `ANSWER` tables. The `ANSWER` tables are not normal database relations, whether permanent or temporary. Their purpose in the query is only to serve as names that are shared among queries and permit coordination. For example, the relation `Reservation` in the example from the introduction is an `ANSWER` relation. There is no relation named `Reservation` in the database; after the queries are evaluated, Kramer and Jerry each receive a result set with the appropriate answer tuple. These answer tuples do not persist anywhere, nor are they accessible to any other queries. In particular, Kramer's answer tuples are not even accessible to Jerry's query and vice versa. The `CHOOSE 1` at the end of the query explicitly specifies that the system should choose exactly 1 tuple among all the tuples which satisfy the coordination constraints, and that such a query should be chosen at random.

This paper presents semantics and an evaluation algorithm for entangled queries that are restricted to use only select-project-join (conjunctive) queries on the `ANSWER` relations in the `WHERE` clause, and arbitrary queries otherwise. Such queries are powerful and expressive enough to handle many real-world coordination scenarios. We discuss potential extensions in Section 6.

2.2 Intermediate representation

Although entangled queries are specified in an extension of SQL, their evaluation is easier to perform on an intermediate representation. The representation uses a Datalog-like syntax; however, it does not involve any recursion and it is completely equivalent to the SQL syntax presented above.

In this representation, an entangled query has the form

$$\{C\}\ H\ :-\ B$$

where C and H are conjunctions of relational atoms over answer relations and B a query over database (non-answer) relations. B, H and C are the *body*, *head* and *postcondition* of the query, respectively. Each atom in the representation may contain constants and variables. All variables that appear in H or C must also appear in B (a range-restriction requirement). For simplicity of discussion, we

$$\{\texttt{R}(\texttt{Jerry}, x)\}\ \texttt{R}(\texttt{Kramer}, x)\ \text{:−}\ \texttt{F}(x, \texttt{Paris})$$

$$\{\texttt{R}(\texttt{Kramer}, y)\}\ \texttt{R}(\texttt{Jerry}, y)\ \text{:−}\ \texttt{F}(y, \texttt{Paris}) \wedge \texttt{A}(y, \texttt{United})$$

(a)

1:	$\{\texttt{R}(\texttt{Jerry}, 122)\}$	$\texttt{R}(\texttt{Kramer}, 122)$
2:	$\{\texttt{R}(\texttt{Jerry}, 123)\}$	$\texttt{R}(\texttt{Kramer}, 123)$
3:	$\{\texttt{R}(\texttt{Jerry}, 134)\}$	$\texttt{R}(\texttt{Kramer}, 134)$
4:	$\{\texttt{R}(\texttt{Kramer}, 122)\}$	$\texttt{R}(\texttt{Jerry}, 122)$
5:	$\{\texttt{R}(\texttt{Kramer}, 123)\}$	$\texttt{R}(\texttt{Jerry}, 123)$

(b)

Figure 2: (a) Intermediate representation of entangled queries (b) Grounded queries

restrict B to conjunctions of relational atoms for the remainder of this paper. This is, however, not enforced by the model in general.

For an entangled query expressed in extended SQL, H corresponds to the `SELECT INTO` clause, while B and C correspond to information in the `WHERE` clause. C specifies all the conditions on answer relations from the `WHERE` clause. B specifies the conditions on database relations from the `WHERE` clause, as well as serving to bind variables used in H and C.

Figure 2 (a) shows the intermediate representation of Kramer and Jerry's queries from the introduction. The relations `Reservation`, `Flights` and `Airlines` are abbreviated as `R`, `F` and `A` respectively.

2.3 Semantics

From the point of view of a single entangled query, evaluation is a process that returns an *answer*, i.e. a single row from the appropriate answer relation. From the point of view of the system, evaluation always involves a set of entangled queries, and the goal is to *populate* the answer relation in a way that respects all queries' coordination constraints. In the running example, Kramer and Jerry wish to coordinate on flight numbers. The system evaluates their queries by finding a tuple for Kramer's query and a tuple for Jerry's query that share the same flight number, and returning each tuple as an answer to the appropriate query.

Consequently, coordination semantics must be defined from the perspective of the system, by specifying how a set of entangled queries must be answered together. The process which the system must perform is called *coordinated query answering*; it is described next. For correctness, it is necessary to ensure that the underlying database is not changed during the answering process.

Grounding the queries: Coordinated query answering makes use of two technical concepts – *valuations* and *groundings*. If q is a query in the intermediate representation and the current database is D, a valuation is simply an assignment of a value from D to each variable of q. For example, on the database in Figure 1 (a), Kramer's query has three valuations: x can be mapped to either 122, 123 or 134. Every valuation of a query is associated with a *grounding*, which is q itself with the variables replaced by constants following the valuation. We use the terms "grounding" and "grounded query" interchangeably.

Let $\mathcal{Q}$ be the set of queries to be evaluated in a coordinated manner. In the description that follows, we make use of $\mathcal{G}$, the set of groundings of the queries on the database. It is important to understand that evaluation does not require that $\mathcal{G}$ be materialized; indeed, our evaluation algorithm presented in Section 4 does not materialize it. However, for the purpose of explaining the semantics, $\mathcal{G}$ is a useful tool.

Figure 2 (b) shows the set $\mathcal{G}$ obtained by grounding Kramer and Jerry's queries on the database in Figure 1 (a). The bodies of the groundings are no longer needed and can be discarded.

Finding the answers: At a high level, the evaluation is a search for a subset $\mathcal{G}' \subseteq \mathcal{G}$ such that $\mathcal{G}'$ contains at most one grounding of each query and the groundings in $\mathcal{G}'$ can all mutually satisfy each other's postconditions. That is, if all the heads of the groundings in $\mathcal{G}'$ were combined into a set, this set would contain all the postconditions. Any set of groundings satisfying this property is called a *coordinating set*. Once such a $\mathcal{G}'$ is found, the evaluation produces an answer relation which consists of the union of all the head atoms in $\mathcal{G}'$ (the answer may consist of more than one relation – this will happen if the head atoms refer to more than one relation, i.e. the original queries mention more than one `ANSWER` relation).

In the example, the initial set $\mathcal{G}$ is as shown in Figure 2 (b). Groundings 1 and 4, as well as groundings 2 and 5, are suitable coordinating subsets $\mathcal{G}'$. Either of them may be used to generate the answer relation and return answers to the respective queries.

It is possible that the selected $\mathcal{G}'$ might not contain any groundings for some queries. This event can be thought of as a statement that those queries could not be answered; it is up to the programmer to determine how to handle this case in the transaction code.

Guarantees on answering: In general, multiple suitable coordinating sets $\mathcal{G}'$ may exist. This raises the question of what requirements one should place on evaluation. It is clearly desirable that some $\mathcal{G}'$ be found unless none exists, and perhaps also that the $\mathcal{G}'$ chosen be maximal, i.e. contain groundings of as many queries as possible. However, the following result exposes fundamental limitations on the guarantees that we can provide efficiently.

Theorem 2.1. *Given a set of queries $\mathcal{Q}$ and an instance of the database D, it is NP-complete to determine whether there exists a coordinating set $\mathcal{G}' \subseteq \mathcal{G}$, where $\mathcal{G}$ is the set of all groundings for $\mathcal{Q}$ on D, containing at most one grounding of each query from $\mathcal{Q}$.*

This result is not surprising, considering that – as discussed – entangled queries are powerful enough to encode instances of CSP.

3. QUERY ANSWERING IN PRACTICE

The main reason for the complexity of entangled query evaluation indicated in Theorem 2.1 is not actually the choice of the data values – such as specific flight numbers or hotel rooms. The complexity is due to the fact that if we consider arbitrary sets of queries, a backtracking search [6, 17] is required to discover the coordination *structure*, that is, the way the queries (and their respective groundings) match up together. Moreover, sometimes this coordination structure is not unique.

Fortunately, real-world users are very unlikely to generate sets of entangled queries that encode complex constraint satisfaction. In fact, the sets of queries that they do generate are likely to have a very specific structure. It turns out that we can put this knowledge to good use in developing an efficient evaluation algorithm. In this section, we formalize this additional structure and explain why it allows tractable evaluation with respect to the data complexity.

3.1 Safe and Unique coordination

We argue that in most practical scenarios, the coordination structure that users express through entangled queries has two formal properties: it is *safe* and *unique*. We informally introduce each of these properties in turn before formalizing them and explaining how they jointly guarantee tractability of evaluation.

We begin with the notion of *safe* coordination. Consider Kramer and Jerry's example queries from our running example. Each query has a clear coordination partner. This means there is one clear desired global outcome: both Jerry and Kramer receive the details of a United flight to Paris. Suppose, however, that we extend the

$$\begin{aligned}
\{\mathtt{R}(\mathtt{Jerry}, x)\}\ \mathtt{R}(\mathtt{Kramer}, x) &\ {:}{-}\ \mathtt{F}(x, \mathtt{Paris}) \\
\{\mathtt{R}(\mathtt{Jerry}, y)\}\ \mathtt{R}(\mathtt{Elaine}, y) &\ {:}{-}\ \mathtt{F}(y, \mathtt{Athens}) \\
\{\mathtt{R}(f, z)\}\ \mathtt{R}(\mathtt{Jerry}, z) &\ {:}{-}\ \mathtt{F}(z, w) \wedge \mathtt{Friend}(\mathtt{Jerry}, f)
\end{aligned}$$

(a)

$$\begin{aligned}
\{\mathtt{R}(\mathtt{Jerry}, x)\}\ \mathtt{R}(\mathtt{Kramer}, x) &\ {:}{-}\ \mathtt{F}(x, \mathtt{Paris}) \\
\{\mathtt{R}(\mathtt{Kramer}, y)\}\ \mathtt{R}(\mathtt{Jerry}, y) &\ {:}{-}\ \mathtt{F}(y, \mathtt{Paris}) \\
\{\mathtt{R}(\mathtt{Jerry}, z)\}\ \mathtt{R}(\mathtt{Frank}, z) &\ {:}{-}\ \mathtt{F}(z, \mathtt{Paris}) \wedge \mathtt{A}(y, \mathtt{United})
\end{aligned}$$

(b)

Figure 3: (a) An unsafe set of queries (b) A set of queries which is not unique

database in our flight booking scenario with a Friend relation, and that three users – Kramer, Jerry and Elaine – are mutual friends. Consider the three queries in Figure 3 (a). The queries represent the fact that Kramer wants to coordinate with Jerry on a flight to Paris, Elaine wants to coordinate with Jerry on a flight to Athens, and Jerry is happy to coordinate with any friend on any flight.

This set of queries does not fully specify the structure of the desired coordination. Jerry's query has two potential queries in the set that could be its coordination partners; however, his query requires a single tuple as an answer. There are two possible coordination outcomes that satisfy *some* users: either Jerry flies with Kramer or he flies with Elaine. However, there is no outcome that satisfies all users, and it is unclear how the system might choose between the two outcomes above.

To understand what it means for a coordination structure to be *unique*, consider the three queries shown in Figure 3 (b). Here Jerry and Kramer wish to coordinate on a flight to Paris as before. In addition, Frank wishes to coordinate with Jerry on a flight to Paris, but only if the airline is United. Depending on the flight database, there are several possibilities for coordination here. First, it may be possible to book all three users on a United flight. Of course, it is possible that no suitable United flights exist. In this case, Jerry and Kramer may still be able to coordinate and fly with another airline. The coordination structure here is safe – each query has a unique coordination partner – but it is not unique. There are proper subsets of the entire set of queries that may be able to coordinate "locally" even if the entire set cannot.

We next formalize the two above notions.

3.1.1 Safety

Formally, a safe set of queries can be characterized in terms of logical unifiability between various head and postcondition atoms of the queries in the set. Consider two relational atoms containing constants and variables that involve the same relation. They are unifiable unless they contain different constants for the same attribute value; for example, $\mathtt{R}(x, y)$ and $\mathtt{R}(z, z)$ are unifiable whereas $\mathtt{R}(2, y)$ and $\mathtt{R}(3, z)$ are not. We call a set of queries Q *unsafe* if it contains a query q with a postcondition atom that is unifiable with two (or more) head atoms found in Q. These can be either head atoms of two different queries, or two head atoms of the same query. Evaluation of such queries is intractable and leads to degradation in the performance of the system.

For example, in Figure 3 (a), Jerry's query has a postcondition atom R(f,z) which unifies with the head of Kramer's query as well as the head of Elaine's query. Therefore, the set of queries is unsafe.

If presented with a set of queries which is unsafe, the system has several options. Ideally, the problem would be pointed out to the users involved and they would receive feedback allowing them to reformulate their queries. Alternately, the system could remove queries from the set until the remaining set was safe. A simple way to do this is to iterate over the query set and search for queries q with postconditions that unify with more than one head atom. All such queries q would be removed from the set when found. This procedure is not in general Church-Rosser, but it is simple and can be performed efficiently. More sophisticated strategies for query removal may be appropriate in particular application settings.

3.1.2 Uniqueness of the coordination structure

The formal definition of safety involves excluding queries whose postconditions unify with more than one head. Uniqueness of the coordination structure, on the other hand, has to do with heads that unify with more than one postcondition, as seen in the three queries in Figure 3 (b): the head atom of the second query, $\mathtt{R}(\mathtt{Jerry}, y)$ unifies with the postcondition atoms of both the first and third query. However, the restriction required for uniqueness of coordination structure (UCS) is not as straightforward as excluding all queries with such heads; sometimes these types of configurations can be permitted. Intuitively, the problem is due to the fact that a subset of the queries can coordinate separately of the rest.

To define the UCS property for a set of queries, we use a simplified version of the unifiability graph that will be introduced in more detail in Section 4. Construct a graph with a node for every query in the system. Draw an edge from node q_i to q_j if a head atom of q_i unifies with a postcondition atom of q_j. Intuitively, if there is a path from query q_k to q_l, this means that groundings of query q_l require groundings of q_k for satisfaction, directly or transitively.

We can use this graph to define UCS. We say that a set of queries has the UCS property if every node in its simplified unifiability graph belongs to a strongly connected component of the same graph. This excludes the type of behavior shown in Figure 3 (b). The simplified unifiability graph for this set of queries has three nodes, one for each query. There are three edges – edges in both directions between Jerry and Kramer's queries, and an additional edge from Jerry's query to Frank's query. Thus, Frank's query does not belong to a strongly connected component of the graph.

An interesting property is that a set of queries could satisfy the UCS property even though a query in the set is unsafe. For example, the third query shown in Figure 3 (a) is part of the strongly connected component of the graph although it is unsafe.

3.2 Tractable evaluation

In settings where the coordination structure is both safe and unique, efficient evaluation is possible.

Theorem 3.1. *If a set of entangled queries Q is safe and UCS, then all the queries can be evaluated in PTIME with respect to data complexity.*

The intuition for why efficient evaluation is possible is that the coordination structure can be discovered efficiently. If we construct a graph based on the unifiability of the head and postcondition atoms of the query, the strongly connected components of the unifiability graph correspond to sets of queries that are coordination partners and require each other's postconditions during evaluation.

Within each such group, the specific way in which the queries match is unique. It is therefore possible to collect the queries together into a big query that specifies a single joint outcome based on the way they match. This is explained in much greater detail in Section 4, but as an example, Jerry and Kramer's queries from the introduction can be combined into this postcondition-free query:

$$\mathtt{R}(\mathtt{Kramer}, x) \wedge \mathtt{R}(\mathtt{Jerry}, x)\ {:}{-}\ \mathtt{F}(x, \mathtt{Paris}) \wedge \mathtt{A}(x, \mathtt{United})$$

This query specifies that the system should find a United flight to Paris and return the two answer tuples to Jerry and Kramer.

In the evaluation process as outlined above, safety guarantees tractability, by ensuring that there is a unique way to combine the queries in each strongly connected component into a bigger query. The UCS property guarantees correctness: we know that we will not miss any possible answers (i.e. coordinating sets of groundings) that involve proper subsets of a set of matching queries, as explained in our discussion of the queries in Figure 3 (b).

4. THE EVALUATION ALGORITHM

We now introduce our algorithm for coordinated query answering. Within our system, this algorithm is implemented in the coordination module as explained in Section 5.1. It is invoked by the coordination middleware, either automatically at regular intervals or through explicit requests. Upon invocation, the algorithm operates on a snapshot of the database and on a fixed set Q of queries. The set Q is assumed to be safe; if necessary, a simple check can be run on Q to ensure safety.

The algorithm has two main phases: query matching and evaluation proper. Query matching discovers the coordination structure implicit in the individual entangled queries and uses this structure to construct a set of combined queries. Once each combined query is available, it is sent to the database for evaluation; each answer to this query corresponds to a set of answers to the individual entangled queries. The first (or any other) combined query answer can be used to produce the individual answers.

4.1 Query Matching

Query matching discovers the coordination structure implicit in the set of entangled queries. In most cases, as discussed, users submit small groups of queries that match only each other. That is, the structure consists of a potentially large number of small, disconnected groups of queries that will coordinate only internally.

The query matching phase discovers this structure in two steps. First, it identifies the disconnected, independent groups of queries. In doing so, it partitions Q into a set of components which can subsequently be processed independently and in parallel. We call this phase the partitioning phase and describe it in Section 4.1.2.

Next, the algorithm works on each group of queries to discover the actual coordination by determining how the query heads and postconditions match. We refer to this phase as matching (proper) and describe it in Section 4.1.3.

All stages of this process make use of a data structure called the *unifiability graph* that represents certain dependencies among the queries in Q with respect to matching. We begin by introducing this graph and explaining how it is constructed. We then discuss how the subsequent phases make use of it.

4.1.1 The unifiability graph

The *unifiability graph* of a set of queries Q is a multi-digraph (directed multigraph) that contains a distinct node $N(q_i)$ for each query q_i in Q. There is an edge from query node $N(q_i)$ to query node $N(q_j)$ for each pair of atoms (h, p) such that h is a head atom of q_i, p is a postcondition atom of q_j, and h unifies with p. For the remainder of this section, we use q_i to represent both a query in Q and the corresponding node in the unifiability graph.

For every query q_i in Q, let $\mathtt{INDEGREE}(q_i)$ denote the indegree of the corresponding graph node, and let $\mathtt{PCCOUNT}(q_i)$ equal the number of postconditions of query q_i. Safety guarantees that there will be at most one edge into a graph node q_i for each postcondition of q_i. This means that for every query q_i in Q,

$$\mathtt{INDEGREE}(q_i) \leq \mathtt{PCCOUNT}(q_i)$$

Equality holds if and only if every postcondition atom of q_i unifies with a head atom of some query.

For instance, suppose Q consists of the three following queries:

$$\begin{aligned} q_1 &: \ \{\mathtt{R}(x_1) \wedge \mathtt{S}(x_2)\}\ \mathtt{T}(x_3) :- \mathtt{D1}(x_1, x_2, x_3) \\ q_2 &: \ \{\mathtt{T}(1)\}\ \mathtt{R}(y_1) :- \mathtt{D2}(y_1) \\ q_3 &: \ \{\mathtt{T}(z_1)\}\ \mathtt{S}(z_2) :- \mathtt{D3}(z_1, z_2) \end{aligned}$$

Then the unifiability graph is as shown in Figure 4 (a). We will use this set of three queries as our running example for this section.

4.1.2 Partitioning

The unifiability graph allows Q to be partitioned into subsets that can be processed separately and in parallel. These partitions are precisely the connected components of the unifiability graph; for convenience, we refer to the queries corresponding to a connected component of the unifiability graph as a *component* of Q. Suppose that queries q_1 and q_2 are in different components of Q. Then any coordinating set that contains groundings of both q_1 and q_2 can be broken into two smaller disjoint coordinating sets, one of which contains q_1 and the other of which contains q_2. All subsequent stages of evaluation can therefore be performed separately on each component of Q. Partitioning the graph has other potential benefits in addition to the performance advantages associated with increased parallelization and smaller search spaces. For instance, it has security benefits. By analyzing the unifiability graph, an implementation of our system could provide guarantees about the interaction between different queries in the system. A system sensitive to privacy could partition the workload by grouping queries into sets of “trusted and sensitive,” “trusted but not sensitive,” or “untrusted” queries and ensure that no component of Q could contain both a “trusted and sensitive” and an “untrusted” query.

4.1.3 Unifier Propagation

At the core of our algorithm is an iterative process that identifies and removes unanswerable queries, i.e. those that have no chance to participate in a coordinating set. Fundamental to the algorithm is the observation that a query with a postcondition that does not unify with any query’s head cannot have a grounding that participates in a coordinating set. Any such query can therefore be safely disregarded. We can identify such queries using our unifiability graph: a query node $N(q_i)$ can be safely removed from the graph if its indegree is strictly less than the number of postconditions of q_i.

Unifier propagation requires that no variable can appear in more than one query. If the initial Q does not satisfy this property, it is easy to enforce it by renaming variables as needed. For the remainder of this section, we assume that each variable is indeed unique to a single query. Let $\mathtt{Val}$ denote the set of all constants and variables occurring in Q.

Unifiers The matching algorithm associates a *unifier* $U(n)$ with each node n in the unifiability graph. A unifier is a constraint on the valuations of the variables in $\mathtt{Val}$. Formally, it is a partition of a subset of $\mathtt{Val}$ which contains at most one constant per partition class. It can be represented as a set of subsets of $\mathtt{Val}$. For example, $\{\{x, 3\}, \{y, z\}\}$ is a unifier specifying that in any permitted valuation, the variable x must have value 3 and the variables y and z must have the same value.

Given unifiers u_1 and u_2, the *Most General Unifier* of u_1 and u_2, denoted $mgu(u_1, u_2)$, is the most general (least restrictive) unifier that enforces all the constraints imposed by each u_i. In general,

$mgu(u_1, u_2)$ may not exist, but if it does exist then it is unique. For instance, there is no most general unifier for the unifiers $\{\{x, 3\}\}$ and $\{\{x, 4\}\}$; if one existed, it would need to restrict valuations so that x was equal to both 3 and 4.

Given two unifiers u_1 and u_2, it is possible to compute $mgu(u_1, u_2)$ – or determine that it does not exist – using standard methods. An optimized implementation of the MGU procedure based on disjoint-set forests provides strong performance guarantees. If unifiers u_1 and u_2 jointly contain k distinct variables then it possible to compute their most general unifier in expected $O(k \cdot \alpha(k))$ time, where α is the inverse of the Ackermann function.

Cascading effects of unifier propagation If a query node q_i is removed from the graph then we can also remove any node q_j such that a postcondition atom of q_j unifies with a head atom of q_i. This is true because of our safety condition: we know that each postcondition atom unifies with at most one head atom.

In practice, this means that if a node q_i is removed from the unifiability graph then every successor q_j of q_i may be removed as well. Repeating this argument, we may remove every successor of a successor of q_i, and so on until we have removed all descendants of q_i from the graph. This can be accomplished using a standard graph traversal algorithm such as Breadth-First Search. We assume that there is a function `CLEANUP`(n) that removes an input node and all its descendants from the dependency graph, as well as all edges into and out of those nodes. We also assume that `CLEANUP` removes all of these nodes from the *updates* queue, a data structure whose purpose will be described shortly.

4.1.4 Matching

We are now ready to explain the query matching algorithm proper.

We begin by constructing a unifiability graph for the set of queries Q. For each query q_i in Q, we create a node, and we define a set $U(q_i)$, called the *unifier*, for this node. Intuitively, $U(q_i)$ represents the minimal (least restrictive) currently known constraints on valuations that must hold for any coordinating set that contains a grounding of q_i.

We initialize the unifier $U(q_i)$ of each node q_i to the empty set. For each head atom h of each query q_i we check whether there is a postcondition atom p of a query q_j that unifies with it. If such a p exists then we create an edge from q_i to q_j in the unifiability graph. We also update $U(q_j)$ to be the MGU of $U(q_j)$ and the most general unifier of p and h. If no such h exists or no MGU exists then the query q_i is unsatisfiable, and we may run `CLEANUP` to remove it and all its descendants from the graph.

The unifiability graph can be generated in a straightforward but inefficient manner by trying to unify each postcondition with each head in our entire input set of queries. This process can be made more efficient by building indices, but doing so is non-trivial. For example, consider the atoms `Reserve (Kramer, x)` and `Reserve (Jerry, y)`. Clearly, a unifier does not exist for these atoms despite the fact that they point to the same relation. Interestingly, we can *attempt* to reduce the number of these matchings by simply replacing the variables in every atom by a unique constant Δ. We then build an index on all heads in Q of the following form:

```
(Relation, Parameter, Value) → [List of Atoms]
```

A lookup for a postcondition atom `Reserve (Jerry, y)` involves a seek on the index for `(Reserve, 1, Jerry)` and `(Reserve, 1, Δ)`. Formally, if $\mathcal{L}$ denotes the lookup function on the index and $\mathcal{A}$ represents the set of atoms, an atom $R(v_1 \ldots v_n)$ can only unify with

$$\mathcal{A} \cap \bigcap_{constants\ v_i} (\mathcal{L}(R, i, v_i) \cup \mathcal{L}(R, i, \Delta))$$

Such an index structure does not provide us with any guarantee on complexity. Indeed, we expect it to perform poorly when queries have many variables. However, a query set with a very large number of variables is highly likely to be unsafe: postconditions and heads that contain mostly variables rather than constants will typically unify with each other densely. In practice, therefore, this type of index is immensely useful.

In building the graph, we iteratively removed any query containing a postcondition that did not unify with some head atom. This fact, together with our assumption that Q is safe, is sufficient to guarantee that that for each postcondition p of each query q_i in the graph there is exactly one other query q_j with a head h that unifies with p. This establishes a local satisfaction of constraints for each of the remaining nodes in the dependency graph. The algorithm next propagates these constraints using the structure of the unifiability graph. More specifically, if a postcondition of query q_j requires the head of some query q_i for satisfaction, the coordinating set cannot contain a grounding of q_j unless both q_j's existing constraints and q_i's constraints hold.

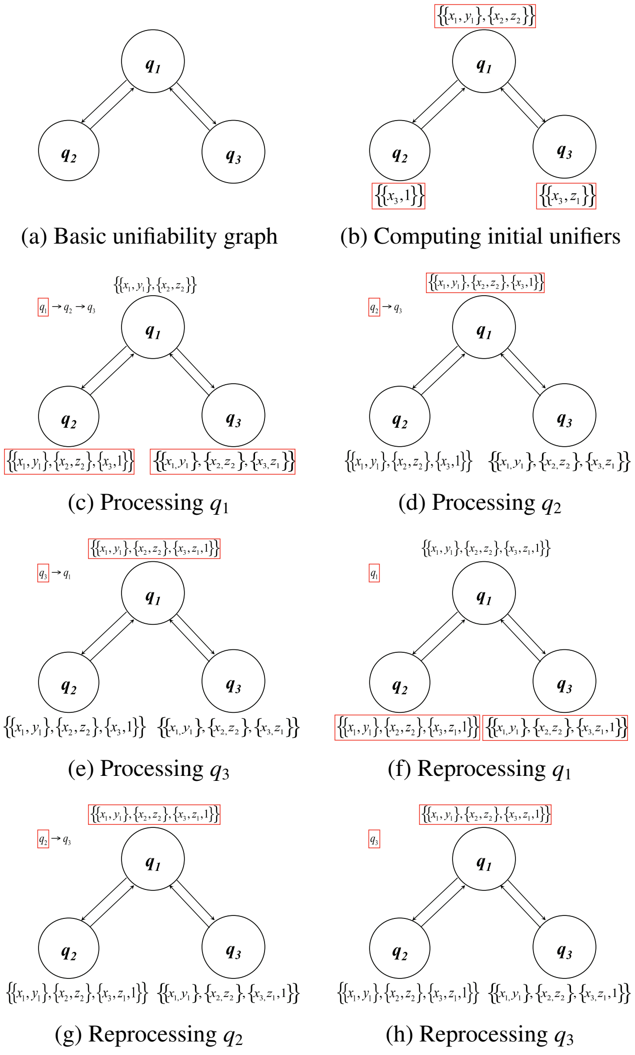
Unifier propagation is an iterative procedure that runs on each component of the unifiability graph. As it runs, it performs two tasks. First, it discovers the coordination structure, i.e. how the queries match with respect to satisfying each other's postconditions. As it does this, it updates the unifiers associated with the graph nodes to reflect the current known constraints on valuations that are required for this query to be answerable. Simultaneously, the algorithm discovers and removes unanswerable queries from the graph.

Algorithm 1 Matching on a unifiability graph G

```
updates := queue containing all nodes in G
while updates is not empty do
  parent := DEQUEUE(updates)
  for child in successors of parent do
    U(child) := MGU(U(parent), U(child))
    if U(child) was changed then
      if U(child) = NIL then
        CLEANUP(child)
      else
        ENQUEUE(updates, child)
      end if
    end if
  end for
end while
```

The propagation procedure is shown in Algorithm 1. At a high level, it pushes unifier information forward along edges. If a unifier does not exist for some node q_i then the `CLEANUP` function is invoked on q_i, removing it and all its descendants from the unifiability graph and the *updates* queue. The intuition is that such a node corresponds to an unanswerable query, and any descendants of this node represent queries that relied on a postcondition of q_i for satisfaction, so are also unanswerable. Whenever the unifier of a node is updated, that node is added to the *updates* queue so that the change can be propagated to the node's children. This propagation of unifier information continues until no new information is propagated by any of the nodes and the *updates* queue becomes empty.

The execution of the algorithm on our running example is shown in Figure 4. In Figure 4 (b), unifiers are computed for all nodes in the graph, and all nodes in the graph are added to the *updates* queue. In Figure 4 (c), the first node, q_1, is removed from the head of the queue and information about its constraints is propagated to


$\{\{x_1, y_1\}, \{x_2, z_2\}\}$
q_1
q_2
q_3
$\{\{x_3, 1\}\}$
$\{\{x_3, z_1\}\}$
$q_1 \rightarrow q_2 \rightarrow q_3$
$\{\{x_1, y_1\}, \{x_2, z_2\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1\}\}$
$q_2 \rightarrow q_3$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1\}\}$
$q_3 \rightarrow q_1$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1\}\}$
q_1
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$q_2 \rightarrow q_3$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
q_3
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$

(a) Basic unifiability graph

(b) Computing initial unifiers

(c) Processing q_1

(d) Processing q_2

(e) Processing q_3

(f) Reprocessing q_1

(g) Reprocessing q_2

(h) Reprocessing q_3

Figure 4: A sample run of matching

its successors q_2 and q_3. In 4 (d), q_2 is removed from the queue and information about its constraints is propagated to its child q_1. Since q_1 is not currently in the queue, it is added at this point. In 4 (e), q_3 is removed from the queue and its constraints are propagated to its child q_1. In 4 (f), q_1 is processed again with its new unifier, and information about the update is propagated to q_2 and q_3. In 4 (g) and 4 (h), the update is propagated to q_1, but since $U(q_1)$ is not changed by the operation, it is not added to the queue.

We now consider a variant of this example in which q_3 has the postcondition $\mathtt{T}(2)$ rather than $\mathtt{T}(z_1)$. In this case, no choice of head atoms for q_1 can simultaneously satisfy the postconditions of q_2 and q_3, so we expect that the matching algorithm should fail. Indeed, immediately before Figure 4 (e), $U(q_2)$ will contain the set $\{x_3, 1\}$ and $U(q_3)$ will contain the set $\{x_3, 2\}$. The unifier of q_1 will be updated first to $mgu(U(q_1), U(q_2))$ and then to the unifier of that value with $mgu(U(q_1), U(q_2))$. The last unification will require x_3 to be equal to 1 and 2 simultaneously, and that unification will therefore fail. As expected, the matching algorithm will consequently eliminate the node q_1 and its children q_2 and q_3.

4.1.5 Complexity Analysis

Graph Construction We first analyze the complexity of constructing the unifiability graph. Let H denote the total number of head atoms in all queries in $\mathcal{Q}$, let P denote the total number of postcondition atoms, and let κ denote the greatest number of columns that appears in any single atom in $\mathcal{Q}$. In the absence of any indices, for each head atom h and postcondition atom p in $\mathcal{Q}$, we must check whether h unifies with p; each such check takes expected $O(\kappa\ \alpha(\kappa))$ time. If h is fixed then we must perform this check with P different values of p. Since every query in $\mathcal{Q}$ contains at least one postcondition atom, the time required to find all postcondition atoms and perform this loop is expected $O(P\ \kappa\ \alpha(\kappa))$. We must perform this inner loop for H different values of h. Since each query in $\mathcal{Q}$ has at least one head atom, finding all the head atoms in the input and iterating over all of them takes expected $O(P\ H\ \kappa\ \alpha(\kappa))$ time.

Unifier Propagation We now analyze the complexity of Algorithm 1. The input is a connected component of the unifiability graph containing nodes $\mathcal{Q}' \subset \mathcal{Q}$ such that each variable appears in at most one query in $\mathcal{Q}'$, as well as a unifier for each node in the graph. Suppose that all queries in the input jointly contain k free variables, and let w be the maximum number of postconditions of any query in $\mathcal{Q}$. Let P be the total number of postcondition atoms in every query in the graph, and n the number of queries in $\mathcal{Q}'$.

We add a node to the *updates* queue only at the very beginning of the algorithm or when its unifier is updated by a call to the MGU function on line 5 of the algorithm pseudocode. First suppose that $k = 0$, i.e. there are no variables in the input. In this case, unification is trivial, unifiers are never changed, and the whole algorithm runs in time proportional to the number of edges in the graph; this is bounded above by $O(P)$ time.

Now suppose that $k > 0$. If a unifier is updated by a call to the MGU function then either the new unifier must contain a constant that the old unifier did not contain or else two sets in the old unifier must be merged together and the total number of sets in the unifier must decrease. This means that if all queries in the input jointly contain k free variables then for each node *child* in $\mathcal{Q}'$, the check on line 6 can succeed at most $O(k)$ times. If every node q in the input has indegree at most w then each node can be added to the *updates* queue at most $O(kw)$ times. It follows that lines 5-12 can be executed at most once $O(kw^2)$ for each node in the graph. Each execution takes expected $O(k \cdot \alpha(k))$ time if we ignore the time spent in the `CLEANUP` function on line 8, so the running time of the loop is expected $O(k^2\ w^2 \cdot \alpha(k))$. The total time spent in the `CLEANUP` function across all calls is at worst linear in the number of nodes in the input. It follows that the entire procedure runs in $O(k^2\ w^2\ \alpha(k) + n + P)$ time. Since every query in the input contains at least one postcondition, this can be simplified to expected $O(k^2\ w^2\ \alpha(k) + P)$ time.

4.1.6 Discussion

We note that at any given time, the unifier of a query node q represents the weakest constraints on variables that must hold in order for there to be a coordinating set of groundings for a subset $\mathcal{Q}' \subset \mathcal{Q}$ that contains exactly one grounding for each query in $\mathcal{Q}'$. A node is removed from the graph only when this is known to be impossible, either because some of its postconditions can't be satisfied at all or because some subset of its postconditions can't be mutually satisfied by any variable assignment.

This is the best we can do without any knowledge of the records in the database: the unifier of any query that remains in the system after the matching algorithm halts can be satisfied for some valuation of the variables it contains. This means that for each remaining query q there exists a database D, a set of groundings $\overline{\mathcal{Q}} \subset \mathcal{Q}'$, and a coordinating set of groundings $\mathcal{G}$, such that $q \in \overline{Q}$ and $\mathcal{G}$ consists of exactly one grounding for each $g \in \mathcal{G}$.

4.2 Constructing and evaluating the combined query

After the matching procedure finishes, we are left with a set of answerable queries $Q = \{q_i\}_{i \in I}$, each associated with a unifier $U(q_i)$, such that Q is a subset of the current component Q' of Q. We compute a global unifier U for the whole set of queries as $mgu(\{U(q_i)\})$. If such a U cannot be computed, evaluation fails for Q' and all the queries in Q' are rejected. If U does exist then it can be expressed as a conjunction of equality statements relating the variables and constants involved; call this conjunction φ_U.

At this point, the evaluation algorithm creates a combined query using Q and φ_U. Let B_i denote the body of query q_i, and let H_i denote the conjunction of its head atoms. Then the combined query q^* is

$$\bigwedge_i H_i \text{ :- } \bigwedge_i B_i \wedge \varphi_U$$

That is, the body of q^* is the conjunction of all the bodies of the original queries, together with equality atoms that encode the constraints in u. The head of q^* is the conjunction of the original query heads.

In our running example illustrated in Figure 4, all query nodes end up with the same unifier after matching. This is

$$\{\{x_1, y_1\},\ \{x_2, z_2\},\ \{x_3, z_1, 1\}\}$$

The required most general unifier U is consequently also

$$\{\{x_1, y_1\},\ \{x_2, z_2\},\ \{x_3, z_1, 1\}\}$$

A suitable corresponding φ_U is

$$x_1 = y_1 \wedge x_2 = z_2 \wedge x_3 = z_1 \wedge x_3 = 1$$

The combined query generated by the system is as follows:

$$\begin{aligned} \mathtt{T}(x_3) \wedge \mathtt{R}(y_1) \wedge \mathtt{S}(z_2) \text{ :- } & \mathtt{D1}(x_1, x_2, x_3) \wedge \mathtt{D2}(y_1) \wedge \mathtt{D3}(z_1, z_2) \\ & \wedge x_1 = y_1 \wedge x_2 = z_2 \wedge x_3 = z_1 \wedge x_3 = 1 \end{aligned}$$

As this example makes clear, q^* can be simplified making use of the information in φ_U. Our example query is equivalent to the following query:

$$\mathtt{T}(1) \wedge \mathtt{R}(x_1) \wedge \mathtt{S}(x_2) \text{ :- } \mathtt{D1}(x_1, x_2, x_3) \wedge \mathtt{D2}(x_1) \wedge \mathtt{D3}(1, x_2)$$

Once q^* is constructed, it can be sent to the database for evaluation. Each answer to q^* is a valuation of the variables in q^* that corresponds to a set of fully grounded head atoms. Only one such valuation is necessary to answer the entangled queries, so q^* may be equipped with a `LIMIT 1` clause. Once an answer is available, the fully grounded head atoms can be used to generate answers for the individual queries from Q in a straightforward manner.

5. D3C ENGINE AND EXPERIMENTS

This section describes the system we are building to provide end-to-end support for entangled queries. We outline the structure of our system and present results from an experimental evaluation of our implementation of the query evaluation algorithm.

5.1 D3C Engine

Designing an entire system to provide end-to-end coordination support is a major research challenge. For instance, all levels of the system must handle not just coordination success, but coordination failure as well. Suppose Kramer submits his query as in our first example, but Jerry's matching query never arrives; the system needs a suitable mechanism for dealing with this, ultimately sending a message to the transaction code that the query is not answerable. As another example, suppose Kramer and Jerry do coordinate, but Kramer's transaction aborts before he makes the booking. The coordination has created a dependency between their transactions which must be considered during recovery. Integrating entangled queries into a transaction processing system is ongoing work.

In [9], we argue that designing for D3C raises questions about the very foundations of database system design. Coordination, by definition, requires communication between user programs. As such, it is a breach of isolation, which is a cornerstone of the transaction abstraction. If transactions are no longer isolated, this has fundamental implications for the overall system architecture at all levels.

Figure 5 gives the outline of the portion of our system which is directly involved in handling entangled queries. The design is closely tied to the life cycle of the entangled queries, from the moment the query is generated until the answers are returned.

Entangled queries can, in principle, be input by hand, but normally they are generated by a front end web interface, just like regular (non-entangled) queries. Once a query is generated, it is passed to a suitable layer for answering.

From the perspective of the application, the coordinated answering is an asynchronous process. An individual query may not in general be answerable until other, partner queries are available. The middleware layer provides to the application an asynchronous query answering abstraction with callback functionality. Such an abstraction is needed due to the misalignment between the asynchronous query *submission* by the application code and the synchronous entangled query *answering* by the coordination module.

It is unrealistic for an entangled query to wait an arbitrary amount of time for a coordination partner. To deal with this, the system has a notion of *query staleness*; when a query becomes stale, it is removed from the list of pending queries and its evaluation is considered to have failed. Any further handling of the query is up to the programmer in application code. Staleness can be defined in a variety of ways; a timeout mechanism or manual user intervention are two possibilities.

Below the middleware layer, a dedicated module actually performs the coordination. The structure of this module directly mirrors that of the algorithm presented in Section 4. It receives a stream of queries and constructs the unifiability graph using suitable indices over the queries. Subsequently, each component of the graph can be processed by an independent server thread, which performs the actual query matching and generates a combined query. This combined query is then sent to the database for evaluation. The DB query optimizer can apply traditional query optimization techniques in evaluating this combined query. Once the coordination module computes answers to the individual entangled queries, these answers are returned back to the application code.

At present, we have a full implementation of the coordination module. The implementation consists of a server which can accept connections and queries from a hundred clients. The evaluation algorithm can be executed periodically in a set-at-a time fashion (after specific time intervals or after a fixed number of queries). Alternately, it can be executed incrementally upon submission of every query. On the arrival of a new query in the system, the unifiability graph may be updated and only certain partitions may require updates. The incremental evaluation requires each partition to store the partial matching unifiers and continues the matching algorithm from this state with the addition of a new query. A parameter in our implementation allows us to switch between the two. Section 5.3.4 discusses the impact of using each of these approaches.

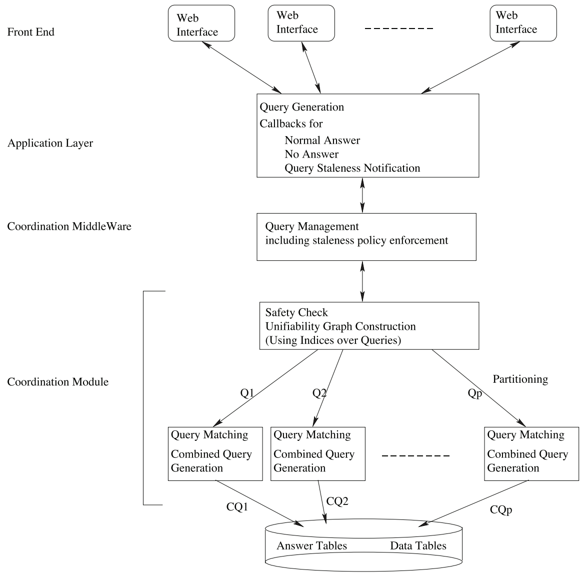


Figure 5: D3C Engine based on entangled queries

The system is implemented in Java 1.6.0. The implementation uses JDBC to connect to a MySQL database system (version 4.1.20).

5.2 Experimental Setup

To evaluate the system, we use a simulated flight booking scenario in which users want to coordinate their travel plans with their friends. We use the *Slashdot* social network data [2] to establish friendship relationships between users. The graph has 82168 users and 102 airport destinations. We assign a "hometown" airport to each of the users, ensuring as far as possible that that each user has at least half his or her friends living in the same city.

The schema for our system is as follows:

```
Reserve(UserName, Destination)
Friends(UserName1, UserName2)
User(UserName, HomeTown)
```

In the rest of this section, we use R, F and U to denote the Reserve, Friends and User tables respectively. Within this flight booking scenario, we test our system under various different coordination scenarios and with different workloads.

We run all experiments on a Dual 2.0Ghz Intel Xeon CPU with 5GB of RAM; the reported values are averages over three runs. The standard deviation is less than 2% in each experiment.

5.3 Results

We present results from five sets of experiments. The first three are designed to test the scalability of coordination in an increasingly complex set of scenarios; the last two stress-test our query matching and safety check procedures. All experiments use an incremental version of the algorithm unless specified otherwise.

5.3.1 Two-way coordination

The first experiment tests the scalability of coordinated query answering in a basic scenario where pairs of friends want to coordinate on flights. The query sets used consist of pairs of queries of the following form:

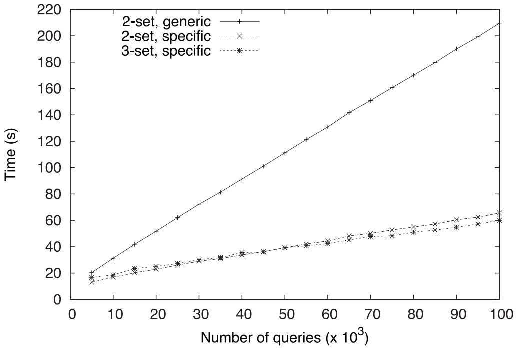


Figure 6: Scalability on best-case and random workload

$$\{\mathtt{R}(x,\mathtt{ITH})\}\ \mathtt{R}(\mathtt{Jerry},\mathtt{ITH})\ :-$$
$$\mathtt{F}(\mathtt{Jerry},x) \wedge \mathtt{U}(\mathtt{Jerry},c) \wedge \mathtt{U}(x,c)$$
$$\{\mathtt{R}(x,\mathtt{ITH})\}\ \mathtt{R}(\mathtt{Kramer},\mathtt{ITH})\ :-$$
$$\mathtt{F}(\mathtt{Kramer},x) \wedge \mathtt{U}(\mathtt{Kramer},c) \wedge \mathtt{U}(x,c)$$

The intuition is that the above pair of queries is generated by Jerry and Kramer who each want to fly to JFK with any of their friends. When generating such query pairs, we ensure that Jerry and Kramer are friends according to the social network structure, but we do not ensure that they live in the same city. Enforcing only one of these two conditions in query generation allows us to produce queries that have a realistic – not too small and not too large – chance to coordinate.

We vary the size of our query sets from five to one hundred thousand. In addition, to detect any side effects of our incremental query evaluation approach, each run of the experiment is evaluated on a randomly permuted set of mutually coordinating pairs of queries. Figure 6 shows our results.

It is interesting to note that although the heads and postconditions of all queries point to the same ANSWER relation, the performance of system is linear in the number of queries. This is due to the fact that queries coordinate often and the number of "pending" queries in the system does not grow with an increase in the number of queries.

We also test the effect of making the queries more specific. In particular, we eliminate the variables from the postcondition and the head, so that the pairs queries are now of the following form:

$$\{\mathtt{R}(\mathtt{Kramer},\mathtt{ITH})\}\ \mathtt{R}(\mathtt{Jerry},\mathtt{ITH})\ :-$$
$$\mathtt{F}(\mathtt{Jerry},\mathtt{Kramer}) \wedge \mathtt{U}(\mathtt{Jerry},c) \wedge \mathtt{U}(\mathtt{Kramer},c)$$
$$\{\mathtt{R}(\mathtt{Jerry},\mathtt{ITH})\}\ \mathtt{R}(\mathtt{Kramer},\mathtt{ITH})\ :-$$
$$\mathtt{F}(\mathtt{Kramer},\mathtt{Jerry}) \wedge \mathtt{U}(\mathtt{Kramer},c) \wedge \mathtt{U}(\mathtt{Jerry},c)$$

Earlier, a join was required in the body between F and U to ground the value of x. However, with the complete specification of friends, this join is now eliminated and the grounding step is faster. This leads to an increase in performance, as shown in Figure 6. The overall performance of the system, however, is still linear in the number of queries.

5.3.2 Three-way coordination

The second experiments tests scalability in a slightly more complex scenario. We now generate triples of queries, corresponding to triangles in the social network structure, of the following form:

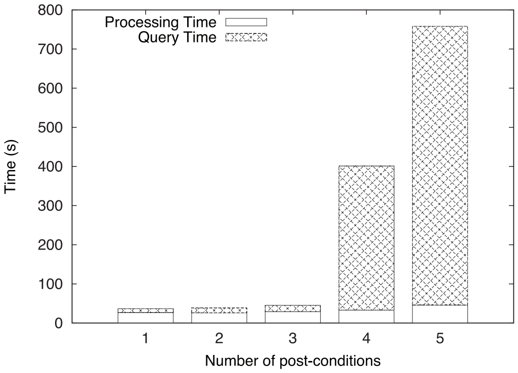


Figure 7: Scalability in the number of postconditions

$$\{\texttt{R(Kramer,IAH)}\}\ \texttt{R(Jerry,IAH)}\ :-$$
$$\texttt{F(Jerry,Kramer)} \wedge \texttt{U(Jerry,}c\texttt{)} \wedge \texttt{U(Kramer,}c\texttt{)}$$
$$\{\texttt{R(Elaine,IAH)}\}\ \texttt{R(Kramer,IAH)}\ :-$$
$$\texttt{F(Kramer,Elaine)} \wedge \texttt{U(Kramer,}c\texttt{)} \wedge \texttt{U(Elaine,}c\texttt{)}$$
$$\{\texttt{R(Jerry,IAH)}\}\ \texttt{R(Elaine,IAH)}\ :-$$
$$\texttt{F(Elaine,Jerry)} \wedge \texttt{U(Elaine,}c\texttt{)} \wedge \texttt{U(Elaine,}c\texttt{)}$$

We vary the size of the query set within the same parameters as before. Figure 6 shows the results.

5.3.3 *Increasing the number of postconditions*

The next set of experiments investigates the performance impact of an increase in the complexity of the coordination required. Specifically, we increase the number of postconditions per query, varying it from one to five. For each individual experimental run, all queries have the same number of postconditions. A sample set of three queries with two postconditions is given below.

$$\{\texttt{R(Jerry,SBN)} \wedge \texttt{R(Kramer,SBN)}\}\ \texttt{R(Elaine,SBN)}\ :-$$
$$\texttt{F(Elaine,Jerry)} \wedge \texttt{F(Elaine,Kramer)} \wedge$$
$$\texttt{U(Kramer,}c\texttt{)} \wedge \texttt{U(Elaine,}c\texttt{)} \wedge \texttt{U(Jerry,}c\texttt{)}$$
$$\{\texttt{R(Elaine,SBN)} \wedge \texttt{R(Kramer,SBN)}\}\ \texttt{R(Jerry,SBN)}\ :-$$
$$\texttt{F(Jerry,Elaine)} \wedge \texttt{F(Jerry,Kramer)} \wedge$$
$$\texttt{U(Kramer,}c\texttt{)} \wedge \texttt{U(Jerry,}c\texttt{)} \wedge \texttt{U(Elaine,}c\texttt{)}$$
$$\{\texttt{R(Elaine,SBN)} \wedge \texttt{R(Jerry,SBN)}\}\ \texttt{R(Kramer,SBN)}\ :-$$
$$\texttt{F(Kramer,Elaine)} \wedge \texttt{F(Kramer,Jerry)} \wedge$$
$$\texttt{U(Jerry,}c\texttt{)} \wedge \texttt{U(Kramer,}c\texttt{)} \wedge \texttt{U(Elaine,}c\texttt{)}$$

This represents a scenario where Elaine wants to travel with both her friends, Jerry and Kramer. Jerry and Kramer have analogous requirements. Note that this is different from the three way coordination mentioned above; cliques in the social graph are required for coordination, rather than just cycles. The intent of the coordination is that they all travel together from the same city to the same destination. Queries with a greater number of postconditions are generated in a similar fashion. Increasing the number of postcondition is associated with an increase in the number of queries that must be matched for successful coordination.

Figure 7 shows two components of the result obtained by executing 10000 queries. The first component corresponds to the time taken by the algorithm to find matching sets of queries, and the second part corresponds to the time taken by the MySQL database for query evaluation. The database performs very poorly when the number of joins surpasses a certain threshold (14). However, the time required to find matching sets of queries is still within reasonable bounds in the number of postcondition atoms.

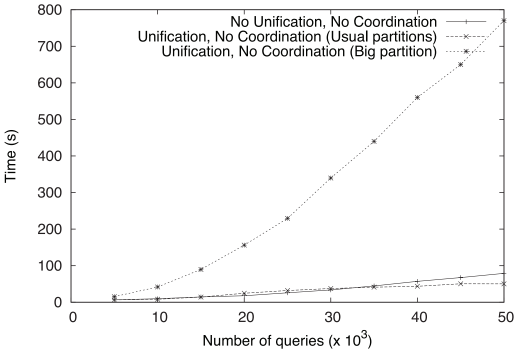


Figure 8: Scalability when queries do not match

5.3.4 *Stress-testing the query matching*

Our next sets of experiments are designed to test the performance of query matching for workloads where little coordination can take place because most queries are unanswerable.

We first test this contingency using a query set generated to ensure that no query has a postcondition unifying with the head of another query. In this case, the unifiability graph does not have any edges; however, with the arrival of each query, index looks are performed to check for new edges. The unifier propagation phase of the algorithm is never initiated because postcondition and head atoms never unify. As expected the "no coordination, no unification" curve in Figure 8 is near-linear.

We also run experiments on a workload in which queries frequently have coordination partners but the system is never able to generate a single combined query in the evaluation phase. This process requires both graph construction and unifier propagation, and ideally the unifier propagation, even for queries without variables, should dominate the running time. If the matching algorithm was run after every query, one would initially expect the algorithm's running time to be at least quadratic.

As the "usual partitions" line in Figure 8 shows, the query evaluation time is near-linear even though there is an increase in the number of pending queries (as no matching takes place) and many queries unify. In other words, the current set of queries forms a long chain in the unifiability graph but does not form cycles. After more careful analysis, we observe that the clustering in the social network graph allows the partitions of the unifiability graph to stay within a certain bound. This explains the high throughput in the experiment on the query set with high unification but no matching.

In order to stress test our system, we identify a big cluster in the social network graph and run experiments on this single large cluster. This change results in significant increase in the overall running time of our experiment. We next run a set-at-a-time evaluation of such massively unifying partitions instead. Figure 8 shows the performance of such a process. It is still within reasonable bounds, given that thousands of people are trying to coordinate together. We therefore establish that for extremely huge coordinating groups, evaluating the queries set-at-a-time is definitely a better approach. By doing so, we wait till all coordination partners arrive before we actually run the algorithm.

5.3.5 *Stress-testing the safety check*

In the final experiment, we test the performance of the safety check. We load the system with twenty thousand queries that are unable to coordinate. Then, we add large sets of queries to the

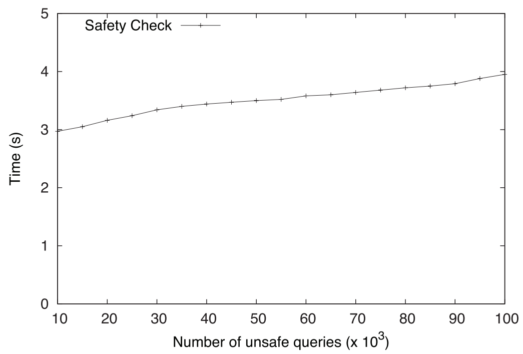


Figure 9: Evaluation time for safety check

system. Such sets contain queries that will fail the safety check with respect to the queries already present in the system. We vary the size of such sets of queries from five to one hundred thousand. The results are shown in Figure 9. It clearly shows that the safety check does not add significant overhead to the system.

5.3.6 Discussion

In designing our experiments, our goal was not to design a full benchmark for entangled queries, but to understand whether this functionality is viable for use in a real-world system. As our results show, the algorithm is efficient in removing queries that are unable to be matched with others and queries that cause safety violations. The queries that are matched can be evaluated efficiently. The overall evaluation algorithm scales to workloads which are realistically sized with respect to today's social networks.

6. FUTURE WORK

Notwithstanding the tractability bounds imposed by Theorem 2.1, a more expressive language for entangled queries would have many practical advantages. In this section, we present several concrete language extensions that would greatly enhance the usefulness of entangled queries. The syntax for entangled queries could be extended with features such as disjunction, union and aggregation in `WHERE` clauses. Consider a database that contains three tables: a table `Parties` with schema `(pid, pdate)`, a table `Friend` with schema `(name1, name2)`, and a relation `Attendance` with schema `(pid, name)`. Suppose a user named Jerry wants to attend a party on Friday subject to the constraint that more than five of his friends attend this same party. This could be expressed as follows using aggregation:

```
SELECT party_id, ‘Jerry’ INTO ANSWER Attendance
WHERE
  party_id IN (SELECT pid
               FROM Parties
               WHERE pdate=‘Friday’)
AND
  (SELECT COUNT(*)
   FROM ANSWER Attendance A, Friend F
   WHERE party_id = A.pid AND
         A.name = F.name2 AND
         F.name1 = ‘Jerry’) > 5
CHOOSE 1
```

"Soft" preferences, another possible extension of entangled queries, would allow coordination constraints to be relaxed when full coordination is difficult. For example, if Jerry and Kramer have trouble obtaining matching travel itineraries, they could instead request that their respective travel dates be as close together as possible.

It is also desirable to allow users to specify a ranking function on preferred query groundings. In our travel example, users who are coordinating on travel dates may prefer some dates to others. Disregarding their preferences may be acceptable if satisfying them precludes coordination, but the evaluation algorithm should favor coordinating sets $\mathcal{G}'$ that satisfy the users' preferences.

Finally, many applications could benefit from extended semantics that allow a query to return more than one answer tuple. Such semantics might allow users to request that *all* groundings of a query be included in the coordinating set, or that as many as possible be included up to some limit k. For instance, in a coordination-aware course enrollment system, students might request that they be enrolled in the same courses as their friends while the registrar ensures that no student enrolls in more than four courses.

Developing these and other extensions fully and designing suitable semantics and evaluation methods for them is ongoing work.

7. ACKNOWLEDGMENTS

This research has been supported by the NSF under Grants IIS-0534404, IIS-0911036, by a Google Research Award, by NYSTAR under Agreement C050061, and by the iAd Project funded by the Research Council of Norway. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

8. REFERENCES

[1] http://zope.stackless.com.

[2] http://snap.stanford.edu/data/soc-Slashdot0902.html.

[3] Web services transactions specification (WS-T). www.ibm.com/developerworks/library/ws-coor/, Aug 2002.

[4] G. Alonso, D. Agrawal, A. E. Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *ICDE*, pages 574–581, 1996.

[5] S. Dalal, S. Temel, M. Little, M. Potts, and J. Webber. Coordinating business transactions on the web. *IEEE Internet Computing*, 7(1):30–39, 2003.

[6] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[7] H. Garcia-Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, 1987.

[8] S. L. P. Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL*, pages 295–308, 1996.

[9] L. Kot, N. Gupta, S. Roy, J. Gehrke, and C. Koch. Beyond isolation: Research opportunities in declarative data-driven coordination. *SIGMOD Record*, 39(1):27–32, 2010.

[10] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2007.

[11] N. A. Lynch and M. Merritt. Introduction to the theory of nested transactions. *Theor. Comput. Sci.*, 62(1-2):123, 1988.

[12] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[13] K. R. Mohan Kamath. Failure handling and coordinated execution of concurrent workflows. In *ICDE*, 1998.

[14] J. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[15] A. Reuter and H. Wächter. The contract model. *IEEE Data Eng. Bull.*, 14(1):39–43, 1991.

[16] J. Roberts and K. Srinivasan. The tentative hold protocol. W3C Note, www.w3.org/TR/tenthold-1/., Nov 2001.

[17] L. Siklóssy and J.-L. Laurière. Removing restrictions in the relational data base model: An application of problem solving techniques. In *AAAI*, pages 310–313, 1982.

[18] R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1996.

[19] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1995.