TRAC: Toward Recency and Consistency Reporting in a Database with Distributed Data Sources *

Jiansheng Huang
University of Wisconsin at Madison
1210 W. Dayton St.
Madison, WI
jhuang@cs.wisc.edu

Jeffrey F. Naughton
University of Wisconsin at Madison
1210 W. Dayton St.
Madison, WI
naughton@cs.wisc.edu

Miron Livny
University of Wisconsin at Madison
1210 W. Dayton St.
Madison, WI
miron@cs.wisc.edu

ABSTRACT

Distributed computing environments, including workflows in computational grids, present challenges for monitoring, as the state of the system may be captured only in logs distributed throughout the system. One approach to monitoring such systems is to "sniff" these distributed logs and to store their transformed content in a DBMS. This centralizes the state and exposes it for querying; unfortunately, it also creates uncertainty with respect to the recency and consistency of the data. Previous related work has focused on allowing queries to express currency and consistency constraints, which are then enforced by "pulling" data from the distributed sources on demand, or by requiring synchronous updates of a centralized data store. In some instances this is impossible due to legacy system issues or inefficient as the system scales to large numbers of processors. Accordingly, we propose that instead of enforcing consistency and recency, such monitoring systems should report these properties along with query results, with the hope that this will allow the data to be appropriately interpreted. We present techniques for reporting consistency and recency for queries and evaluate them with respect to efficiency and precision. Finally, we describe our prototype implementation and present experimental results of our techniques.

1. INTRODUCTION

Grid computing [6] is an umbrella concept for technologies that enable the sharing of computing resources, perhaps even across organizational boundaries, to make computing pervasive and inexpensive. Currently there is a dramatic growth in both the number of processors in grids and the number of jobs users submit to these grids. For reasons of flexibility, scalability, and reliability, the job scheduling and execution systems that run jobs in grids are structured as a collection of independent processes running on the processors of the grid. To aid in debugging, to establish a historical record, and to expose their state to the rest of the system, these processes typically log status records to files on the processors on which they run. This means that the operational and historical data generated as the system executes user jobs are scattered about the grid in these log files. Unfortunately, the result is a nightmare for an administrator trying to ascertain the state of the system or a user trying to understand the status of her jobs.

One promising approach to address this problem is to "sniff" these logs, extract and format the information therein, and load it into an RDBMS. This requires no or minimal changes to the existing job scheduling and execution system, and centralizes and exposes the distributed state of the system to declarative SQL querying, giving users and administrators an easy way to answer their questions. There is intense interest among the grid user community and system adminstrators in having such a tool. However, once one drills deeper into the logistics of such a system, a problem becomes evident: the database will be updated at unpredictable intervals, as each processor in the grid will write to its logs at different rates and times, and each "sniffing" process may make progress at different rates in loading the data into the database. In extreme cases such as machine failures, a node may not "report in" for a long time. This means that the central database will always have an inconsistent view of the system.

Note that this inconsistent view will arise even when the system is running exactly as designed. For example, suppose a user submits a job j to machine m_1, and the job scheduling system decides to run j on a different machine m_2. Depending upon the order in which m_1 and m_2 write their logs and these logs get propagated to the DBMS, we could see at least four different states in response to DBMS queries:

1. Neither m_1 nor m_2 have reported in anything about j.
2. m_1 has reported that j has arrived and has been sent to m_2 to run, but m_2 has not reported receiving j.
3. m_1 has not reported any information about j, but m_2 has reported that it is running j.
4. m_1 has reported that it has received j and sent it to m_2, and m_2 has reported that it is running j.

Even in simpler cases it may be hard for users to interpret the results of their queries. For example, if a user asks "how

*This work was supported by National Science Foundation Award SCI-0515491

VLDB '06, September 12-15, 2006, Seoul, Korea.

many CPU seconds have my jobs used" they may get different answers depending upon which machines have "reported in" to the DBMS.

The standard DBMS approach to resolving this problem would be to insist that the system do everything transactionally. In such an approach, no event (for example, job submission, job commencing execution, job suspension, and so forth) would be allowed to occur without being synchronously logged in the DBMS. In cases where machines of the grid communicate, one would need distributed transactions — for example, in the example of the preceding paragraph, machines m_1 and m_2 would have to participate in a distributed protocol with each other and the DBMS to make sure that only scenarios 1 or 4 are visible to queries.

Unfortunately, this transactional approach is infeasible for several reasons. First, grid job schedulers are large legacy systems and it would be a daunting task to add synchronous distributed transactions everywhere they are needed to guarantee consistency. Second, even if it were feasible to rewrite these systems, the resulting synchronous system would have undesirable blocking behavior (especially when machines fail) and would likely not scale to the ten- or hundred-thousand node grids that are envisioned in the near future. Also, such a synchronous, rigid approach is at odds with the general philosophy of grid systems, in which machines can come and go, jobs and system processes fail regularly, yet the job execution system is flexible and resilient enough to take evasive action and eventually complete the jobs.

Finally, and perhaps most importantly, a transactional view of the distributed system may not be what users want; in many instances what they want is the most recent data available rather than some transactionally consistent view that might ignore the most recent updates from some data sources. For example, in the preceding scenario, a user may prefer to see that job j is running on machine m_2 even if m_1 has not yet reported its submission rather than seeing an earlier report from m_2 that omits j.

Accordingly, in this paper we consider a radically different approach: rather than enforcing transactional consistency, we propose that the system provide recency information along with the answers to user queries. Thus users will still see inconsistent views of the distributed system (this is unavoidable in an asynchronous distributed system that makes data available to users as soon as possible), but they will be able to correctly interpret the answers to their queries despite this inconsistency. For example, in the previous scenario of job j being submitted to m_1 and running on m_2, a user might see in a query response that j is running on machine m_2 despite the fact that it has apparently never been submitted; however, the concerned user could easily determine that this is because m_2 has reported in more recently than m_1.

A naive way to provide such information would be to maintain a table in the DBMS that records, for each data source, the time of the latest update from that data source, and then to tag all data updates with the updating source. This table can be viewed as a vector of "last report times"; we could just return this table along with user queries. A moment's reflection shows that this may be suboptimal. First, how should we maintain consistency between this table and the result of user queries? Second, how should we interpret the lack of a report from a data source? (Is it in trouble, or does it have nothing to report?) Finally, and most interestingly, for many queries we can prove that only a few data sources could possibly impact the query result. In such cases users will suffer unnecessary information overload if we just hand them the whole vector of report times. As an example of this last point, suppose in a ten thousand node cluster a user asks "what jobs do I have running on machine 257?" The user will likely not appreciate an answer that includes a list of the last report times for 9,999 processors in addition to machine 257.

In this paper we propose an alternative approach to providing recency information for such scenarios. Our main contributions are

1. We establish requirements for query-centric data source recency and consistency reporting;
2. We precisely define the concept of which data sources are "relevant" to a given query;
3. We describe how to automatically generate, from a user query, a corresponding recency query, and prove that the resulting query never omits a "relevant" data source;
4. We present a prototype implementation in a monitoring system [9] for Condor [17] and explore, through an experimental evaluation, the impact of our techniques for recency on system performance.

While our target application is job scheduling and execution systems for computational grids, we think these techniques may find broader application. Roughly speaking, the approach of reporting recency rather than enforcing consistency appears useful in systems where a distributed collection of data sources are reporting their state to a centralized DBMS and for which it is infeasible or undesirable to insist on synchronous distributed snapshot. Other examples of such systems include distributed workflows in distributed service oriented architectures and certain categories of sensor networks.

2. RELATED WORK

Early work [1, 7, 12] in replica management and distributed databases allowed local copies of objects to diverge from the master copy and studied various maintenance strategies to guarantee divergence bounds under differing requirements. In data warehousing, [16] introduced query-centric currency driven materialized view refreshing. In web views, [11] defined data freshness metrics and tackled the online view selection problem in the presence of this data freshness information, while [2] used client tunable latency-recency parameters and heuristic functions to decide whether to use a cached object or download a fresh object. More recently, in database caching, [8] explored how to express "good enough" currency and consistency constraints in SQL and how to enforce them during query evaluation.

A common theme in all this previous work is that it enforces recency constraints through a combination of choosing the correct version of an object to query (that is, the cached copy or the primary copy) or refreshing "stale" objects by synchronously "pulling" new data in response to a query. This approach is not viable in our environment, where the DBMS has no choice but to query its (potentially out of date) copy of an object, and where it cannot synchronously "pull" data from the processes being monitored.

The data source recency problem we address is similar to one that arises in data warehousing when multiple potentially remote sources feed into the warehouse. The environments we are targeting (for example, a computational grid with machines in different administrative domains) differ from typical data warehousing in that we have no control whatsoever over the data sources, so that having many sources arbitrarily out of date is "business as usual" rather than an exceptional event to be avoided or flagged or rectified. Also, to our knowledge the published literature on data warehousing does not consider the problem we address: providing query-centric data recency reports along with query results.

Also in the context of data warehousing, [4] investigated how to identify the set of source data items that produced a view item. Their work differs from ours in that they wanted to trace the lineage of a specific data item, while we want to find the recency of the data sources that could possibly impact a given query result. Their lineage-based approach can indeed be modified to determine a subset of the sources that impacted a given query result; however, theirs and other similar lineage-based approaches are not complete, as they can only provide information about data that is present in answer, and may miss sources that impacted the answer by *not* contributing any result tuples.

In the context of distributed databases, [13] addressed the problem of finding the minimal set of locations sufficient to process a query. Their problem statement relies on data items being placed in a distributed environment in such a way that they satisfy various predicates, and then they use the interaction of the data placement predicates and query predicates to identify where data satisfying a simple query might be located. We have a simpler model (where updates are tagged with data sources, and these source tags have special semantics) that allows us to handle more general classes of queries. Finally, our problem is at some level related to the partition pruning techniques implemented in the context of parallel DBMS [5, 10, 14], although in the published literature these techniques focus on noticing when a selection predicate matches the "partitioning predicate" used to allocate data to processors, which again is not sufficient for our purposes.

3. BACKGROUND AND DEFINITIONS

3.1 Terminology

In this paper, the term "data source" is an abstraction that may comprise a monitoring process, the application processes being monitored, and perhaps other processes and files used for communicating data between them. From the point of view of the DBMS, each update is tagged with the time of the event recorded in the update and updates stream in from the source in the order of these timestamps. The details of how these updates get from the application process to the DBMS may vary. For example, it may be that the application process generates data and writes them to a well-known place where the monitoring process will read and report them to the centralized database. The database never "pulls" data from a monitoring process.

A database is a collection of system and user relations. For simplicity, we do not consider user relations that are not updated by the data sources we are monitoring. We assume each user relation receives updates from one or more sources and each tuple is inserted or updated by a single data source.

With each data source the DBMS associates a "recency" timestamp, which represents the most recent timestamp before which all data generated by the application on the data source are guaranteed to be reported to the database. The exact protocol for maintaining the recency timestamp may vary from system to system and is not the focus of this paper. A simple way to do this is to maintain for each data source the timestamp of the most recent event reported by that source. This has the advantage that it does not require any modifications to an application that is already writing logs of events; it has the disadvantage that if the application has nothing to report for a long time it will appear to be a very out of date data source.

It is possible to enable more accurate views of the recency of a source even if it has nothing to report. One way is to require that the application periodically communicate in a "heartbeat" fashion to the corresponding monitoring process, even if it has nothing to report (perhaps by writing a "nothing to report" record with a timestamp to its log.) Finally, in this paper, we assume data is written to reliable storage and that we use a reliable transport mechanism, so that no data is lost from the time it is generated to the time it is reported to the database.

3.2 Guiding Requirements

Our first requirement is concerned with consistency between the user query result and the recency information about the query result. In our paper this recency information is obtained by issuing a system-generated "recency query" along with every user query. If the underlying DBMS uses multiversion concurrency control (MVCC), the consistency constraint means that the same snapshot should be used for both the user query and the corresponding recency query. For lock-based concurrency control, a transaction with serializable isolation-level should be used to guarantee transactional consistency between the two queries.

The second requirement is the completeness of the computed set of "relevant" data sources. Recall that in general our recency report will only cover a subset of the (perhaps thousands) of data sources in the system. This requirement means that no data source that could potentially impact a query's result should be missing from the recency report for that query.

The third requirement is the reverse of the second, and deals with the precision of the reported set of "relevant" sources. Specifically it states that while observing the completeness requirement we should try to minimize the number data sources we report as "relevant" that are not actually relevant. Including "false positive" sources could result in information overload on the user and may even cause the user to take "incorrect" action, for example, waiting for a source to report in when in fact there is no reason to wait.

3.3 Schema Model

Here we clarify our assumptions about the schema the DBMS uses to store the state of the distributed system it is monitoring. Specifically, the schema model needs to take care of two issues: 1) how to model the relationship between data instances and data sources; 2) where to keep track of the recency timestamp of each data source. There are of course many options for how to do this; in the following we discuss one reasonable way and for clarity we use it in the

rest of the paper.

We first consider how to maintain the recency of each data source. For simplicity and efficiency, we want to keep one copy of the recency of each data source. We do this by maintaining a system `Heartbeat` table with two columns: a data source id, and a recency timestamp. The data source column is the primary key in this table. We assume that every contributing data source in a system has an entry in the `Heartbeat` table.

To maintain the association between data sources and rows in the database, we assume that each tuple in a relation is associated with a data source. (Recall that for simplicity we are ignoring tables that are not updated by the data sources being monitored.) This may be achieved by directly adding a data source column to a table, and using this column as a foreign key into the `Heartbeat` table. We expect that often a relation will already contain a column that identifies the data source for each tuple. In this case this existing column can be treated as the data source column. Given a data source 's', we assume that only updates from 's' can insert or change tuples with 's' in the data source field.

3.4 Problem Definitions

We now turn to define precisely what we mean by the set of "relevant" data sources for a query. In order to define what it means for a data source to be "relevant" to a query, we consider two cases separately: 1) a query references one relation; 2) a query references multiple relations. In this paper we assume that a query contains only a single SPJ expression. We first introduce some useful notation:

Notation 1. We use Q to denote a query. For a single-relation query, we use R to denote the relation and use $< c_1, c_2, ..., c_k, c_s >$ to denote the columns of the relation. We use c_s to denote the **data source column** and refer to other columns as **regular columns**. A tuple instance is denoted as $< v_1, v_2, ...,v_k, s >$. The correponding domains for the columns are denoted $D_1, D_2, ..., D_k, D_s$. Specifically, D_s is the domain of the data source column. D_s contains the same set of data source ids that the `Heartbeat` table records.

Notation 2. For a multi-relation query, we use R_i to denote a relation and use $< c_1^i, c_2^i, ..., c_k^i, c_s^i >$ to denote the columns for that relation. We use c_s^i to denote the **data source column** of R_i and call all the other columns **regular columns**. A tuple instance in relation R_i is denoted $< v_1^i, v_2^i, ..., v_k^i, s^i >$. The correponding domains for the columns of R_i are denoted as $D_1^i, D_2^i, ..., D_k^i, D_s$. For simplicity we will also use R to refer to a relation in a multi-relation query when doing so doesn't cause any confusion.

Definition 1. If Q references R, we say that a data source $s \in D_s$ is **relevant** for Q if $\exists v_1 \in D_1, v_2 \in D_2, ..., v_k \in D_k$ s.t. the tuple $< v_1, v_2, ..., v_k, s >$ satisfies Q's predicates.

As we can see, a data source is relevant to a query if it is potentially associated with a tuple that satisfies the query. Note that the tuple doesn't have to exist in the relation. This is because a data source is relevant if it is possible that an update from that data source could change the query result, not just if it appears in a tuple in the query result. Next we turn to the definition of relevance for a query referencing multiple relations.

Definition 2. For a query Q referencing relations $R_1, R_2, ..., R_n$, we say that a data source $s \in D_s$ is **relevant** for Q if $\exists i$ $(1 \le i \le n)$, $\exists v_1^i \in D_1^i, v_2^i \in D_2^i, ..., v_k^i \in D_k^i$, and for $\forall j$ $(j \ne i, 1 \le j \le n)$, $\exists$ a tuple $< v_1^j, v_2^j, ..., v_k^j, s^j > \in R_j$, s.t. the tuple $< v_1^i, v_2^i, ..., v_k^i, s >$ for R_i, $< v_1^j, v_2^j, ..., v_k^j, s^j >$ for each R_j $(j \ne i, 1 \le j \le n)$ together satisfy the predicates of Q. In this case we say that s is **relevant** for Q **via** R_i.

When a query references multiple relations, a data source is relevant if potentially it is associated with a tuple of a relation that joins with the remaining relations and satisfies the other predicates. Here a subtlety is that existing tuples (instead of potential tuples) of the remaining relations are used for joining. If we used potential tuples (as in the single relation case) we might get the unfortunate situation that *all* data sources are always relevant. One reasonable question is whether our definition corresponds to any intuitive user-level guarantee.

It turns out that it does — a common property of the definitions for both the multi-relation and single relation cases is that no single update from an irrelevant data source can change the result of a query. It is possible that an update from a relevant source could change the result. In the multi-relation case, it is also possible that a sequence of changes from an irrelevant data source could change the result. For example, one update from an irrelevant data source may make that source relevant, then another update could change the query result. Note also that the multi-relation definition defaults to the single relation definition if $n = 1$. We formally prove a theorem to make this guarantee shortly after we introduce the set of relevant data sources below.

Notation 3. We denote the defined set of relevant data sources for a query Q as $S(Q)$ and use $A(Q)$ to represent the set of relevant data sources computed by an algorithm.

$$S(Q) = \{s \in D_s | s \; is \; relevant \; for \; Q\}$$

We say that an answer $A(Q)$ is an **upper bound** if $A(Q) \supseteq S(Q)$ and an answer $A(Q)$ is the **minimum** if $A(Q) = S(Q)$.

THEOREM 1. *Let Q reference relations $R_1, R_2, ..., R_n$. We denote the result of Q as $Q(R_1, R_2, ..., R_n)$. If $s \notin S(Q)$, then $\forall i (1 \le i \le n)$, $\forall v_1^i \in D_1^i$, $\forall v_2^i \in D_2^i$, ..., $\forall v_k^i \in D_k^i$, with t_i denoting the tuple $< v_1^i, v_2^i, ..., v_k^i, s >$, the following holds:*

$$Q(R_1, ..., R_i, ..., R_n) = Q(R_1, ..., R_i \cup \{t_i\}, ..., R_n)$$

Informally, this says that if t_i is an update from a data source that is not relevant, it in isolation cannot change the answer to Q.

PROOF. Because Q (as an SPJ expression) can be formulated as a cross product followed by selection and projection, it is not hard to prove that $Q(R_1, ..., R_i \cup \{t_i\}, ..., R_n) = Q(R_1, ..., R_i, ..., R_n) \cup Q(R_1, ..., \{t_i\}, ..., R_n)$. With this we only need to prove that $Q(R_1, ..., \{t_i\}, ..., R_n) = \varnothing$. Assuming $Q(R_1, ..., \{t_i\}, ..., R_n) \ne \varnothing$, then for $\forall j$ $(j \ne i, 1 \le j \le n)$, $\exists$ a tuple $< v_1^j, v_2^j, ..., v_k^j, s^j > \in R_j$, s.t. the tuple t_i for R_i, $< v_1^j, v_2^j, ..., v_k^j, s^j >$ for each R_j $(j \ne i, 1 \le j \le n)$ together satisfy the predicates of Q. By Definition 2, the data source s in t_i is relevant for Q via R_i. This is a contradiction to the condition that $s \notin S(Q)$. □

A final subtlety here is that these definitions talk about the existence of tuples selected from the cross product of the domains of the columns of a relation in a query. If we are also given constraints for the relation schemas, then not all such tuples are valid as updates to the relation instances in question. If constraints are in form of predicates, we can take a user query and append the conjunction of predicates defining such constraints. This converts Q to an equivalent expression Q'. Our definitions can be modified to support such constraints by replacing Q with Q'. For scenarios with constraints not in form of predicates such as key constraints, the definitions of "relevant sources" would have to be augmented to restrict the tuples considered to be those that, when appended to the relation instance, give a legal instance of the relation. This will have the effect in some cases of further increasing the precision of the set of relevant sources identified for a query. If we ignore key constraints on the relations mentioned in a query, it is possible that we overestimate the set of relevant data sources (we say that sources are relevant when actually they are not), because our definitions allow sources to be made relevant by tuples that will never actually occur in the database. Because of this overestimate, we never miss a relevant data source.

Accordingly, in the next section we first ignore such concerns, and leave the development of recency information in the presence of key constraints as an interesting area for future work.

4. TECHNIQUES

In this section we propose techniques for computing the set of data sources that are relevant for a given query. We first show that computing the minimal set of data sources relevant to a query is NP-hard. Despite this negative result, we give efficient algorithms for computing relevant data sources. We prove that they are correct in that they never fail to report a relevant data source; furthermore, we prove that they return the minimum except in two extreme cases: 1) when the user's query is unsatisfiable (it will return no data for any legal instance of the database), or 2) when the user query contains what we call a "mixed predicate" that compares a data source column to a regular column. Even in these extreme cases our techniques may return minimal relevant sets, but we lose the minimality guarantee. Because we suspect these extreme cases are not likely to occur in practice, the NP-hardness result, while theoretically interesting, is not likely to limit the utility of our technique in practice.

4.1 Computing Relevant Data Sources

The definitions in Section 3.4 imply an idea for how to compute the relevant data sources for a query. Taking a single-relation query, for example, we could generate a new relation that is the cross product of the domains for all columns of the relation filtered by constraints, if any. Then we apply the predicates of the query to the new relation by brute force and project out only the data source column. By definition this will produce the minimum answer. While this approach is conceptually simple, it is impractical for two reasons: 1) the domains of some columns may contain an infinite number of values; 2) even if domains of all columns are finite, the performance is likely to be unacceptable because of the size of the cross product. In the following we show that the problem of determining the minimal set of relevant data sources for a query is NP-hard in general.

THEOREM 2. *Given a query Q referencing R, the problem of computing $S(Q)$ is NP-hard.*

PROOF. Assuming the domain of data sources D_s has only one value s, P is the predicates of Q. Let $P(c_s = s)$ be the remaining predicates after we substitute c_s with value s in P. Under the assumptions we make here, the problem of computing $S(Q)$ is equivalent to answering whether s is relevant for Q. Furthermore, if s is relevant for Q, by definition $P(c_s = s)$ must be satisfiable. On the other hand, if s is not relevant for Q, by definition $P(c_s = s)$ can not be satisfiable. Therefore we have reduced the problem of determining the satisfiability of $P(c_s = s)$ to the problem of computing $S(Q)$. Because the satisfiability of $P(c_s = s)$ is NP-hard [15], so is the problem of computing $S(Q)$. □

In the following our approach is to derive constraints on the data source column from the predicates of a query to compute the minimum or an upper bound of the set of relevant data sources. We also present conditions for when the minimum can be reached with a theoretical guarantee.

A query's predicates can be formed using any number of logical operators and comparison operators. To solve the problem uniformly, we first convert the predicate of a query to disjunctive normal form (DNF), which is a disjunction consisting of one or more conjunctive predicates. That is, a query's predicates can be transformed into the following form: $P_1 \vee P_2 \vee ... \vee P_k$, where each P_i is a conjunction of one or more smaller terms, which we call **basic terms**, that are free of $\wedge$ or $\vee$ operators.

COROLLARY 1. *Let Q be a query with predicates in DNF: $P_1 \vee P_2 \vee ... \vee P_k$ where each P_i $(1 \leq i \leq k)$ is a conjunction of basic terms. If Q^1 is the same as Q except with only P_1 as predicates, Q^2 with P_2, ... Q^k with P_k, then*

$$S(Q) = \bigcup_{1 \leq i \leq k} S(Q^i)$$

The proof is evident by applying the definition of a relevant data source to both sides of the equation above. With Corollary 1, we can focus on queries with conjunctive predicates of basic terms. Once again we proceed by first treating single-relation queries, followed by multiple-relation queries.

4.1.1 Single-relation Queries

We first introduce some more notation to facilitate the description and proof of theorems in this section.

Notation 4. Let P be the predicates of Q. We separate P into three parts, $P_s \wedge P_r \wedge P_m$, where each part is a conjunction of zero or more basic terms such that: each term of P_s references only c_s (the data source column), each term of P_r references only regular columns, and lastly each term of P_m references both c_s and at least one regular column. We call P_s **data source only predicates**, P_r **regular column only predicates** and P_m **mixed predicates**. If there is no basic term for a part, we say it is $NULL$.

Notation 5. We use H to represent the `Heartbeat` table and use $< c_s, c_t >$ to denote its columns. c_s is the data source column and c_t is the recency timestamp. If Q references R, we use P'_s to stand for the predicates after $R.c_s$ is replaced with $H.c_s$ in P_s.

THEOREM 3. *If Q references R, and then predicates of Q are organized as $P_s \wedge P_r \wedge P_m$, and if P_m is NULL and P_r is satisfiable in $D_1 \times D_2 \times ... \times D_k$, then*

$$S(Q) = \pi_{c_s}(\sigma_{P'_s}(H))$$

PROOF. If $s \in S(Q)$, then $\exists v_1 \in D_1, v_2 \in D_2, ..., v_k \in D_k$ s.t. the tuple $< v_1, v_2, ..., v_k, s >$ satisfies the predicates of Q. Consequently the tuple must satisfy partial predicates of Q, i.e., $P_s\ (R.c_s = s)$ = TRUE. By replacing the variable $R.c_s$ with $H.c_s$ in P_s, we have $P'_s\ (H.c_s = s)$ = TRUE. Therefore $s \in \pi_{c_s}(\sigma_{P'_s}(H))$.

On the other hand, if $s \in \pi_{c_s}(\sigma_{P'_s}(H))$, then $s \in D_s$ and $P'_s\ (H.c_s = s)$ = TRUE. By replacing the variable $H.c_s$ with $R.c_s$, we have $P_s\ (R.c_s = s)$ = TRUE. Furthermore since P_r is satisfiable in $D_1 \times D_2 \times ... \times D_k$, implying $\exists v_1 \in D_1$, $v_2 \in D_2$, ..., $v_k \in D_k$ s.t. the tuple $< v_1, v_2, ..., v_k >$ satisfies the P_r. Therefore the tuple $< v_1, v_2, ..., v_k, s >$ satisfies P_s , P_r and P_m (which is NULL), by Definition 1. we have $s \in S(Q)$. □

Intuitively Theorem 3 means that if all predicates of a query can be separated into data source only predicates and regular column only predicates, and if the regular column only predicates are satisfiable, then we can directly apply the data source only predicates as constraints on the `Heartbeat` table to get the set of relevant data sources for the query. The set of data sources found through this approach is the minimum.

There are two cases which would break the assumptions of Theorem 3: 1) there are mixed predicates, 2) the regular column only predicates are not satisfiable. The first case occurs when a user wants to compare the data source column to some other column of the same relation. The second case happens when a user specifies contradictory predicates within a query or in a more likely scenario specifies predicates contradictory to constraints. When predicates are not satisfiable, a straighforward result can be obtained as in the following.

COROLLARY 2. *Let Q reference R and the predicates in Q be P. If P is not satisfiable in $D_1 \times D_2 \times ... \times D_k \times D_s$, then the following holds:*

$$S(Q) = \varnothing$$

For all cases where the minimum is not guaranteed either because satisfiability of predicates is unknown or there are mixed predicates, our solution always guarantees an upper bound with Corollary 3.

COROLLARY 3. *Suppose Q references R and that the predicats in Q can be organized as $P_s \wedge P_r \wedge P_m$. The following holds:*

$$S(Q) \subseteq \pi_{c_s}(\sigma_{P'_s}(H))$$

The first part of the proof for Theorem 3 proves Corollary 3.

We introduce the following example for illustration. Suppose we keep track of the activities on machines across administrative boundaries in a single table:

`Activity(mach_id, value, event_time)`

The attributes of the `Activity` table are machine ID, activity value and the event time when an activity value becomes valid. We treat the machine ID as the data source column. Table 1 shows an example instance of the `Activity` relation. Now suppose that a user would like to know which of m_1 and m_2 have reported an "idle" state, and does so with the following query Q_1:

Table 1: Sample data set for `Activity`

mach_id	value	event_time
m_1	idle	03/11/2006 20:37:46
m_2	busy	02/10/2006 18:22:01
m_3	idle	03/12/2006 10:23:05

```
SELECT mach_id FROM Activity
WHERE mach_id IN ('m1', 'm2') AND value = 'idle';
```

In this example, `mach_id IN ('m1','m2')` of Q_1 is a predicate on the data source column, and `value = 'idle'` is a predicate on a regular column. It is satisfiable because `'idle'` is contained in the domain for `value`. Therefore according to Theorem 3, `mach_id IN ('m1', 'm2')` can be directly applied to the data source domain to find the set of relevant data sources, which are {'m_1', 'm_2'}.

4.1.2 Multi-relation Queries

For multi-relation queries, we need to further break down the relevant data sources in the following way:

$$S(Q, R) = \{s \in D_s | s \text{ is relevant for } Q \text{ via } R\}$$

COROLLARY 4. *If Q references relations R_1, R_2, ..., R_n, then*

$$S(Q) = \bigcup_{1 \le i \le n} S(Q, R_i)$$

The proof of Corollary 4 is also evident by applying Definition 2 to both sides of the equation. With Corollary 4, we can now focus on solving each individual $S(Q, R_i)$. In the declaration of the following notation, we call a predicate a "selection predicate" if it only references columns of one relation and a "join predicate" if it references columns of more than one relation.

Notation 6. We organize predicates of Q involving a relation R_i into five parts: $P^i_s \wedge P^i_r \wedge P^i_m \wedge J^i_s \wedge J^i_{rm}$. Each part is a conjunction of zero or more basic terms. Each term of P^i_s is a selection predicate referencing only $R_i.c_s$. Each term of P^i_r is a selection predicate referencing only R_i's regular columns. Each term of P^i_m is a selection predicate referencing both $R_i.c_s$ and at least one of R_i's regular columns. Each term of J^i_s is a join predicate referencing only c_s in terms of R_i's columns. Each term of J^i_{rm} is a join predicate referencing at least one regular column of R_i and it may or may not reference $R_i.c_s$. We refer to P^i_s as the **data source only selection predicates**, P^i_r as the **regular column only selection predicates**, P^i_m as the **mixed selection predicates**, J^i_s as **data source only join predicates**, J^i_{rm} as **regular column only or mixed join predicates**. We use P^i_o to denote all the other predicates of Q excluding the ones referencing R_i.

Here we do not distinguish regular column only join predicates and mixed join predicates. We will explain the reason shortly after Theorem 4.

Notation 7. If Q references R_1, R_2, ..., R_n, we use $P_s^{i'}$ to denote the predicates after $R_i.c_s^i$ is replaced with $H.c_s$ in P_s^i, and $J_s^{i'}$ to stand for the preciates after $R_i.c_s^i$ is replaced with $H.c_s$ in J_s^i.

THEOREM 4. *Suppose Q references relations R_1, R_2, ..., R_n, and that as above, the predicates of Q involving R_i are organized as $P_s^i \wedge P_r^i \wedge P_m^i \wedge J_s^i \wedge J_{rm}^i$. P_o^i includes all the other predicates of Q on the other relations. If P_m^i and J_{rm}^i are NULL and P_r^i is satisfiable in $D_1^i \times D_2^i \times ... \times D_k^i$, then $S(Q, R_i) =$*

$$\pi_{c_s}(\sigma_{P_s^{i'} \wedge J_s^{i'} \wedge P_o^i}(H \times R_1 \times ...R_{i-1} \times R_{i+1}... \times R_n))$$

PROOF. if $s \in S(Q, R_i)$, then $\exists v_1^i \in D_1^i$, $v_2^i \in D_2^i$, ..., $v_k^i \in D_k^i$, and for $\forall j (j \neq i, 1 \leq j \leq n)$, $\exists < v_1^j, v_2^j, ..., v_k^j, s^j > \in R_j$, s.t. the tuples $< v_1^i, v_2^i, ..., v_k^i, s >$ for R_i, $< v_1^j, v_2^j, ..., v_k^j, s^j >$ for R_j $(j \neq i, 1 \leq j \leq n)$ together satisfy all the predicates of Q. Consequently the tuples satisfy $P_s^i \wedge J_s^i \wedge P_o^i$. By replacing the variable $R_i.c_s^i$ with $H.c_s$ in P_s^i and J_s^i, we have s for $H.c_s$, $< v_1^j, v_2^j, ..., v_k^j, s^j >$ for R_j $(j \neq i, 1 \leq j \leq n)$ satisfying $P_s^{i'} \wedge J_s^{i'} \wedge P_o^i$, therefore $s \in \pi_{c_s}(\sigma_{P_s^{i'} \wedge J_s^{i'} \wedge P_o^i}(H \times R_1 \times ...R_{i-1} \times R_{i+1}... \times R_n))$

On the other hand, if $s \in \pi_{c_s}(\sigma_{P_s^{i'} \wedge J_s^{i'} \wedge P_o^i}(H \times R_1 \times ...R_{i-1} \times R_{i+1}... \times R_n))$ then for $\forall j (j \neq i, 1 \leq j \leq n)$, $\exists < v_1^j, v_2^j, ..., v_k^j, s^j > \in R_j$, s.t. s for $H.c_s$ and the tuple $< v_1^j, v_2^j, ..., v_k^j, s^j >$ for R_j $(j \neq i, 1 \leq j \leq n)$ satisfy $P_s^{i'} \wedge J_s^{i'} \wedge P_o^i$. By replacing the variable $H.c_s$ with $R_i.c_s^i$ in $P_s^{i'}$ and $J_s^{i'}$, we have s for $R_i.c_s^i$ and $< v_1^j, v_2^j, ..., v_k^j, s^j >$ for R_j $(j \neq i, 1 \leq j \leq n)$ satisfy $P_s^i \wedge J_s^i \wedge P_o^i$. With the condition that P_r^i being satisfiable in $D_1^i \times D_2^i \times ... \times D_k^i$, there must $\exists v_1^i \in D_1^i, v_2^i \in D_2^i, ..., v_k^i \in D_k^i$, s.t. the tuple $< v_1^i, v_2^i, ..., v_k^i, s >$ satisfies P_r^i. Therefore the tuples $< v_1^i, v_2^i, ..., v_k^i, s >$ for R_i, $< v_1^j, v_2^j, ..., v_k^j, s^j >$ for R_j $(j \neq i, 1 \leq j \leq n)$ satisfy $P_s^i \wedge P_r^i \wedge J_s^i \wedge P_o^i$, which are all the predicates of Q since P_m^i and J_{rm}^i are $NULL$. By Definition 2. $s \in S(Q, R_i)$. □

Therefore the relevant data sources of Q via R_i can be computed by a semijoin between the `Heartbeat` table and the other relations when the conditions are met. Like the single relation case, the assumptions of Theorem 4 will break when there are mixed selection predicates (the P_m^i part) or when there is a contradiction in P_r^i. One additional case for multi-relation query is when there are regular column only or mixed join predicates (the J_{rm}^i part). Such join predicates involve at least one regular column. In this case, the cross product of the data source domain and the domains of any regular column referenced in the predicates can be joined with the other relations to further filter out any irrelevant data sources. This again would be problematic if the domain for a column is infinite or the cross product of the domains is large. The problem applies regardless whether we have regular column only join predicates or mixed join predicates. This is the reason that we do not distinguish them earlier in our notation. Like the single-relation query, we provide an upper bound that is complete for cases where the minimum can't be guaranteed.

COROLLARY 5. *Suppose that Q references relations R_1, R_2, ..., R_n, and that as above, the predicates of Q involving R_i are organized as $P_s^i \wedge P_r^i \wedge P_m^i \wedge J_s^i \wedge J_{rm}^i$. P_o^i includes all the other predicates of Q on the other relations. Then the following holds: $S(Q, R_i) \subseteq$*

$$\pi_{c_s}(\sigma_{P_s^{i'} \wedge J_s^{i'} \wedge P_o^i}(H \times R_1 \times ...R_{i-1} \times R_{i+1}... \times R_n))$$

The first part of the proof for Theorem 4 suffices to prove Corollary 5. Similarly a straightforward result can be obtained when predicates of Q are not satisfiable.

COROLLARY 6. *Suppose that Q references R_1, R_2, ..., R_n and predicates of Q are P. If P is not satisfiable in $D_1^j \times D_2^j \times ... \times D_k^j \times D_s$ $(1 \leq j \leq n)$, then the following holds:*

$$S(Q) = \varnothing$$

We will now give a multi-relation query example to illustrate Theorem 4. Consider a P2P job scheduling and execution system in which each machine in a grid can send jobs to any of its neighbors for execution. Also, suppose that the "neighbor" relationship is captured in the table `Routing(mach_id, neighbor, event_time)` with the semantics that if (m_1, m_2, t) appears in the routing table, at time t m_2 became a neighbor of m_1. We treat the machine ID as the data source column. Table 2 shows a sample instance of the `Routing` relation.

Table 2: Sample data set for `Routing`

mach_id	neighbor	event_time
m_1	m_3	03/12/2006 23:20:06
m_2	m_3	02/10/2006 03:34:21

Now suppose a user would like to know which of the neighbors of m_1 have reported an "idle" state, so the user issues the following query Q_2.

```
SELECT A.mach_id
FROM Routing R, Activity A
WHERE R.mach_id = 'm1' AND A.value = 'idle' AND
R.neighbor = A.mach_id;
```

By applying Corollary 4, we have $S(Q_2) = S(Q_2, R) \bigcup S(Q_2, A)$. To compute $S(Q_2, R)$, we identify `R.mach_id = 'm1'` as a data source only selection predicate (i.e., the P_s), `R.neighbor = A.mach_id` as a regular column only join predicate (i.e., the J_{rm}) and `A.value = 'idle'` as the remaining predicate (i.e., the P_o) on the other relation. Because J_{rm} is not $NULL$, we can't apply Theorem 4 to guarantee the minimum. However, with Corollary 5, we can get

$$S(Q_2, R) \subseteq \pi_{H.c_s}(\sigma_{H.c_s='m_1' \wedge A.value='idle'}(H \times A))$$

Because the value 'idle' appears in the `Activity` table, the expression in the above evaluates to {'m_1'}.

If we assume the domain of `Routing.neighbor` is the same as the domain for `Activity.mach_id`, the regular column only join predicate `R.neighbor = A.mach_id` is also satisfiable in this example. Therefore in fact the upper bound found above is the minimum even though Corollary 5 doesn't guarantee it in general. An extreme counter-example is when the domain of `Activity.mach_id` has no intersection with the domain of `Routing.neighbor`. In this case J_{rm} always evaluates to $FALSE$. Therefore $S(Q_2, R) = \varnothing$, which is a proper subset of the upper bound.

Now let us turn to the problem of finding $S(Q_2, A)$. Because `A.value = 'idle'` (i.e., the P_r) is satisfiable and P_m

and J_{rm} are all $NULL$, Theorem 4's assumptions all hold true. Therefore we have $S(Q_2, A) =$

$$\pi_{H.c_s}(\sigma_{R.neighbor=H.c_s \wedge R.mach\_id='m_1'}(H \times R))$$

Given our data, the right side evaluates to {'m_3 '}.

The example above can be modified slightly to illustrate the point that a sequence of updates from an irrelevant data source can change the result of a query. If we change the sample instance for the `Activity` table such that all three machines have 'busy' value, then by evaluating the same expressions above we will get $S(Q_2, R) = \varnothing$ and $S(Q_2, A) = \{'m_3'\}$. This means that no single update from m_1 or m_2 can change the query result. Now suppose there are two updates in order from m_1: 1) m_1 is updated to 'idle' in `Activity` and 2) m_1 is added as a neighbor of m_1 itself in `Routing`. The first update will make m_1 relevant via `Routing`. The second update will change the query result to include m_1. An interesting observation here is that this particular scenario would not occur if we had an explicit constraint on the `Routing` table that a machine can't have itself as a neighbor.

4.2 Impact of Query Semantics on Recency

A subtle issue is that in such database queries which look very similar may produce different result and recency, between which we observe that there is a tradeoff. Suppose in an example computing system, jobs are submitted to machines which are called scheduling machines. For simplicity let us assume that the scheduling machine for a job decides where it is executed. Assume the database schema for the system has two tables as following:

```
S(schedMachineId, jobId, remoteMachineId)
R(runningMachineId, jobId)
```

The idea is that S captures what the scheduler thinks is happening, R captures what the running machine thinks is happening. Whenever a scheduler assigns a job to a machine, or changes the machine for a job, it updates its tuple for that job to reflect the change. Whenever a running machine is running a job, it reports that fact. R and S are supposed to capture the current state, but they can allow inconsistencies due to time lags.

Now suppose there is a user who wants to know "is my job, which I submitted to the scheduling machine myScheduler, running yet?"

This user may write one of two different queries identified by Q_3 and Q_4 respectively:

```
SELECT R.runningMachineId FROM R
WHERE R.jobId = myId;

SELECT R.runningMachineId FROM S, R
WHERE S.schedMachineId = myScheduler AND
      S.jobId = myId AND R.jobId = myId AND
      R.runningMachineId = S.remoteMachineId;
```

Both capture the user's idea, but they have different semantics (Q_3 does not require a scheduling machine tuple) and different recency.

1. Without any of our techniques, for both Q_3 and Q_4 we would report that all sources are relevant.

2. With our techniques, for Q_3, we will report that all machines are relevant. If any running machine has reported myId, then the query will return the ID for that machine, else nothing.

3. With our techniques, for Q_4, the answer depends upon what we find in S:

   (a) If there is nothing in S for myId and myScheduler, the query will return an empty result and we will report that only myScheduler is relevant; this is correct because only updates from myScheduler can change the query result.

   (b) If there is a tuple in S for myId and myScheduler, but it doesn't join with anything in R, then the query will return an empty result and we will report that myScheduler and S.remoteMachineId are relevant. This is correct because only updates from these two machines can change the answer.

   (c) If there is a tuple in S for myId and myScheduler, and it joins with a tuple in R, then the query will return R.runningMachineId and we will report that myScheduler and R.runningMachineId are relevant.

4.3 Reporting recency and consistency

Getting the set of data sources relevant to a user query solves only one part of the puzzle. In this section we use those results to query the recency timestamps for the relevant data sources and compute descriptive statistics that indicate the overall recency and consistency of a user query result. By "consistency" we mean the relative recency of a set of data sources.

In our approach, the recency timestamps for the relevant data sources of a query are stored in an automatically created temporary table, which is a snapshot of recency information transactionally consistent with the user query result. The temporary table persists until the end of a user session. The user can decide whether to copy it to a permanent table before the end of a session or to allow it to be discarded automatically by the system. With the detailed recency information in a temporary table, users and applications are able to query them to look into recency information of relevant data sources that is consistent with a query result.

When there are only a few relevant data sources, it may be sufficient for users to look at the entire set of returned recency timestamps. However, this will not be an option when there are a large number of relevant data sources. To handle this, we automatically compute the following descriptive statistics to extract some salient features of the set of recency data: the minimum recency timestamp, the maximum recency timestamp, and the range of recency timestamps. The range descriptor in statistics is defined as the difference between the maximum and minimum data points. Other statistics could be computed as well, but we think these three are perhaps the most useful. Specifically, the minimum recency timestamp provides a consistent snapshot for all data sources because all events with timestamps before it must have been reported from all sources. The range of recency timestamps can be interpreted as a bound of inconsistency among the relevant data sources of a query.

In a loosely coupled environment, from time to time we might have data sources that are extremely out of date (for example, when they are suffering from a "hard" network disconnect or failure.) When this happens, if the recency

timestamps of these sources are included in our descriptive statistics they will not be descriptive of the majority of the data sources relevant to a query. To help address this problem, we propose that the system automatically identify "exceptional" data sources and report them in a separate temporary table. Then the usual descriptive statistics would be computed over the remaining "normal" data sources.

Obviously there are many methods that could be used to define what counts as an "exceptionally out of date" data source. One reasonable approach that has found acceptance in other domains is the z-score [3] method from statistics. The z-score is an indicator of how far an individual observation is from the mean of a data set. The idea of using z-score for detecting outlier is based on the Chebyshev theorem [3] which states that for any data set at least 89% of the values will have a z-score less than 3 in absolute value. If the relative frequency distribution of a data set is bell-shaped, then the empirical rule [3] tells that near 100% of the values will have a z-score less than 3 in absolute value.

For each recency timestamp x, the z-score can be calculated with the following formulas:

$$\frac{x-\mu}{\sigma} \; where \; \mu = \frac{\sum_{i=1}^{N} x_i}{N} \; and \; \sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i-\mu)^2}{N}}$$

The μ and σ are respectively the mean and the standard deviation of a data set. x_i represents the recency timestamp of a data source and N is the total number of data sources.

5. PROTOTYPE AND EVALUATION

In this section we address the question of whether our techniques have anything to contribute over the naive approach of simply reporting the recency of all sources along with the answer to user queries. This naive approach has the advantage of simplicity; the reader may wonder if our more complicated approach introduces unacceptable overhead when compared to the naive approach, or if it really succeeds in significantly limiting the size of the reported relevant sources. Our conclusion from running example queries in our prototype implementation is 1) the overhead of our approach is acceptable (in fact, somewhat counter-intuitively, in general our approach has lower overhead than the naive approach), and 2) on the example queries we consider, our technique often succeeds in dramatically reducing the number of sources reported.

5.1 Prototype And Issues

We chose PostgreSQL as our DBMS platform because of its support for table functions. In our prototype, the recency reporting functionality is implemented in a table function that runs the user query and also computes the recency and consistency information, returning them in a temporary table that can be queried subsequently.

Leveraging a table function solves the consistency of recency information and user results in a transparent way. By exposing the reporting mechanism through a table function, users are able to invoke it in one SQL statement within which both the recency query and the original user query are evaluated. By default, PostgreSQL starts a transaction implicitly for each statement and while querying a database each transaction sees a consistent snapshot of the data. Thus consistency between the user query and the corresponding recency query are guaranteed in our prototype.

Suppose a user wants to find all the machines that have reported an "idle" state with the following query over the sample data for **Activity** presented in the example Q_1:

```
SELECT mach_id, value FROM Activity A
WHERE value = 'idle';
```

To get recency information with our techniques, a user can run the query through our prototype function called "recencyReport". In response, the user will get a report (formatted) as in the following. Notice that the "mydb=#" is the prompt from the SQL client.

```
mydb=# SELECT * FROM recencyReport($$
            SELECT mach_id, value FROM Activity A
            WHERE value = 'idle'$$)
           AS t(mach_id TEXT, activity TEXT);
NOTICE:  Exceptional relevant data sources and
         timestamps are in the temporary table:
         sys_temp_e1142376455
NOTICE:  The least recent data source:  m1,
         2006-03-15 14:20:05-06
NOTICE:  The most recent data source: m3,
         2006-03-15 14:40:05-06
NOTICE:  Bound of inconsistency: 00:20:00
NOTICE:  All ‘‘normal’’ relevant data sources and
         timestamps are in the temporary table:
         sys_temp_a1142376455

 mach_id | activity
---------+----------
 m1      | idle
 m3      | idle
(2 rows)

-- query the exceptional relevant data sources
mydb=# SELECT * FROM sys_temp_e1142376455;
 sid |    recency timestamp
-----+-----------------------
 m2  | 2006-02-10 17:23:00-06
(1 row)

-- query the ‘‘normal’’ relevant data sources
mydb=# SELECT * FROM sys_temp_a1142376455;
 sid |    recency timestamp
-----+-----------------------
 m1  | 2006-03-15 14:20:05-06
 m3  | 2006-03-15 14:40:05-06
 m4  | 2006-03-15 14:21:05-06
 m5  | 2006-03-15 14:22:05-06
 m6  | 2006-03-15 14:23:05-06
 m7  | 2006-03-15 14:24:05-06
 m8  | 2006-03-15 14:25:05-06
 m9  | 2006-03-15 14:26:05-06
 m10 | 2006-03-15 14:27:05-06
 m11 | 2006-03-15 14:28:05-06
(10 rows)
```

Besides the normal query result that m_1 and m_3 have an "idle" state, our table function reports that except the relevant data source m_2, all relevant data sources have reported since "2006-03-15 14:20:05-06" and the bound of inconsistency for all such data sources is 20 minutes.

We have encountered two main issues during the course

of implementation. The first issue is that parsing the SQL statement outside the DBMS is not easy, because parsing requires not only a SQL language parser but also access to the schema definition stored in the DBMS so that identifiers can be correctly resolved. The second issue is with the PL/pgSQL, which we used for implementing the table function, is not the ideal language in which to implement a parser. Our current implementation has around 760 lines of PL/pgSQL code, of which more than 2/3 deal with parsing a user query string and generating new query strings. Based on these issues and the system nature of recency and consistency reporting, we think that it would be preferable to implement our functionality within database system instead of as an external add-on.

5.2 Metrics And Evaluation

To measure precision, we calculate the percentage of irrelevant data sources reported vs. the relevant data sources. We call this percentage the **false positive rate (fpr)**. The smaller the false positive rate, the better. Given a query Q, the false positive rate is: $\frac{|A(Q)-S(Q)|}{|S(Q)|}$.

The false positive rate would be impossible to calculate if we did not have a way to determine the relevant data source set for a query. To solve this, we used a test schema specially designed so that a finite domain with a reasonable cardinality is associated with each column of a relation. This way we can apply the brute force idea briefly discussed in the first paragraph of Section 4.1 to determine the relevant data source set for a query. We emphasize that we used this approach only to compute the exact relevant source set in order to analyze our results, not in our recency table function.

To test efficiency, we measure the **response time overhead** caused by the additional recency and consistency reporting. Given a query Q, if $t_1(Q)$ is the response time for evaluating Q and $t_2(Q)$ is the response time for evaluating Q with consistency and recency reporting, the overhead is given by: $\frac{t_2(Q)-t_1(Q)}{t_1(Q)}$.

For our experiment we used Tao Linux 1.0 as the OS on top of a 2.4 GHZ Intel Pentium with 512MB memory. We used PostgreSQL 8.0.0 as the database. The shared buffer pool size was set to 8MB and the size of working memory used for sorting and hash joining was set to 1MB. We used the same schema as given in our previous examples. In addition, we created B-tree indexes on the data source columns of the `Heartbeat`, `Activity` and `Routing` tables.

The data for the tables were synthetically generated and designed to help us understand how the performance overhead scales with respect to the number of data sources and the average amount of data per source. We fixed the total number of rows in the `Activity` table at 10,000,000. Then we varied both the number of data sources and data ratio with inverse proportion. Here the data ratio refers to the number of rows per data source in the `Activity` table. Specifically, we increased the data ratio from 10 to 1,000,000 by factors of 10, while decreasing the number of data sources from 1,000,000 to 10 by the same factor.

The size of the `Activity` table was about 666 MB. The largest `Heartbeat` and `Routing` both had 1,000,000 rows, which was about 58 MB and 74 MB respectively. The largest size of the index was around 303 MB for the `Activity` table and 30 MB for the other two tables. The tables were all analyzed by the PostgreSQL statistics gathering utility before test queries are run.

We have designed four typical test queries for the purpose of demonstrating the overhead of our techniques and precision of our method compared to the naive mathod. The first query accesses a single relation `Activity` using a very selective predicate. The second query does the opposite by using a predicate that is not selective. The third query joins the `Routing` table and `Activity` tables with a very selective predicate on the `Routing` table. The fourth query differs from the third one by using a non-selective predicate on the `Routing` table.

```
Q1: SELECT COUNT(*) FROM Activity A
    WHERE A.mach_id IN ('Tao1','Tao10','Tao100',
          'Tao1000','Tao10000','Tao100000')
          AND A.value = 'idle';

Q2: SELECT COUNT(*) FROM Activity A
    WHERE A.mach_id NOT IN ('Tao1','Tao10',
          'Tao100','Tao1000','Tao10000',
          'Tao100000') AND A.value = 'idle';

Q3: SELECT COUNT(*) FROM Routing R,Activity A
    WHERE R.mach_id IN ('Tao1','Tao10','Tao100',
          'Tao1000', 'Tao10000', 'Tao100000')
          AND R.neighbor = A.mach_id
          AND A.value = 'idle';

Q4: SELECT COUNT(*) FROM Routing R,Activity A
    WHERE R.mach_id NOT IN ('Tao1','Tao10',
          'Tao100','Tao1000','Tao10000',
          'Tao100000') AND R.neighbor = A.mach_id
          AND A.value = 'idle';
```

In the experiments, in addition to measuring our method, which we refer to as the *Focused* method, we also measured the *Naive* method, which is implemented as another table function that queries all data sources in addition to running a user query , and compare their response time overheads and false positive rates for the four test queries. The response times for the *Focused* method includes three parts: 1) the time to parse a user query and generate a recency query, 2) the time to compute relevant data sources using the generated recency query, 3) the time to detect exceptional data sources and compute the least and most recent data sources. The response times for the *Naive* method include the same last two parts as above, but not the first part because it uses a default recency query that covers all data sources. Lastly, we also measure the *Focused* method without the query parsing and generation cost by hardcoding a recency query in the table function. This helps us understand how much overhead comes from the query parsing and generation for the *Focused* method.

For each data set generated, we ran the four queries, their corresponding queries with *focused* and *naive* recency and consistency reporting. The response times were collected for all queries and used to compute the overhead of each method. Each individual query was run 11 times and the average response time of the last 10 runs is used to minimize fluctuation. Figure 1 shows the performance overhead curves of the methods respectively for each query.

For queries that are not selective ($Q2$ and $Q4$), the overheads for all methods approach 0% when the data ratio is 100. This is because non-selective user queries will touch

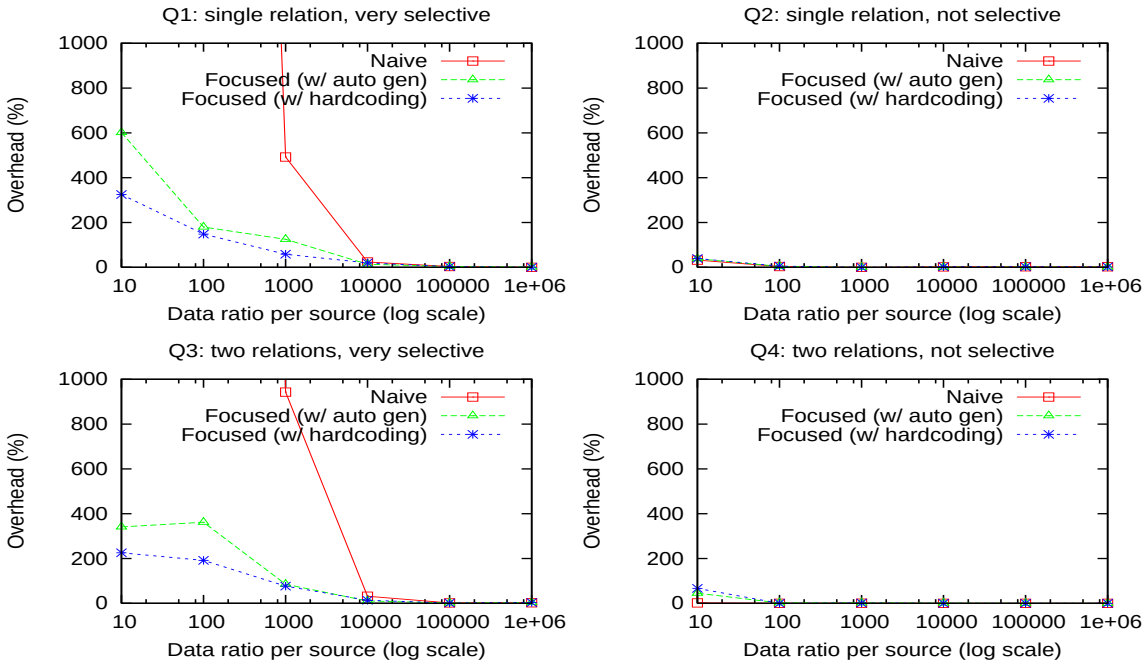

Figure 1: Performance overhead for recency and consistency reporting w.r.t data ratio and # of data sources ((data ratio) × (# of data sources) = 10,000,000).

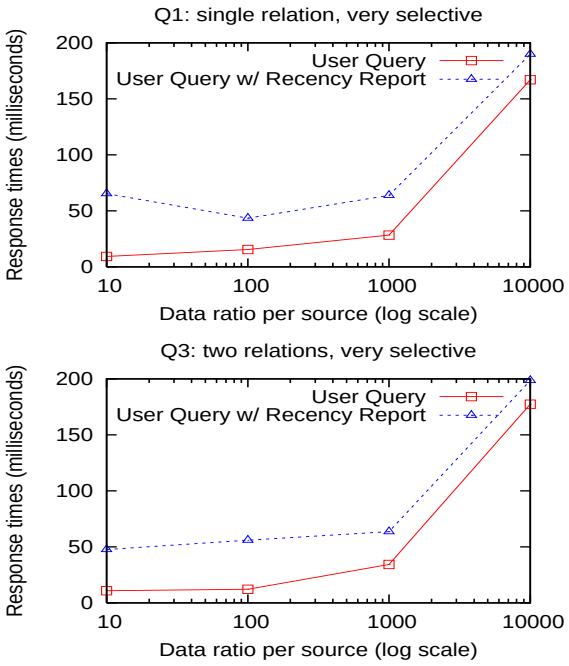

Figure 2: Response times for $Q1$ and $Q3$ with and without recency report w.r.t data ratio and # of data sources ((data ratio) × (# of data sources) = 10,000,000). The *Focused* method with auto generation of recency query is used here.

data of almost all data sources, in which case the data ratio is effectively the factor by which a user query accesses more data than the recency queries. We regard these results as encouraging, because we expect that in most scenarios we would expect each data source to have contributed far more than 10 rows. For the very selective queries ($Q1$ and $Q3$), we see that the overheads approach 0% when the data ratio is 10000. If we zoom into to the portion where the overheads are high (in Figure 2), we discover the reason to be that the user queries have very short running times when the data ratio is 10000 or less. For the same reason, the query parsing and recency query generation also add significant overhead to the *Focused* method. This is also due to the fact that PL/pgSQL processes each expression as a SQL statement sent to the SQL execution engine.

Looking at the other direction where the number of data sources increases, the overhead of the *Naive* method grows rapidly as the number of data sources increases when a user query is very selective. This is because the *Naive* method always queries all the data sources, even if a user query is able to restrict to a few data sources with a very selective predicate. On the contrary, the overhead of the *Focused* method scales much better because it utilizes the selective predicate to probe only relevant data sources.

The only case where the overhead of the *Focused* method is noticably higher than the *Naive* method is when the data ratio is low for $Q4$. The query joins the `Activity` table with the `Routing` table, which is associated with a non-selective predicate. In this case, the *Focused* recency query is a union of two recency subqueries, one for each relation joined. The recency subquery for the `Activity` table also contains a join between the `Routing` table and the `Heartbeat` table. This subquery is almost as expensive as the user query because the data ratio is low. The other recency subquery for the

`Routing` table is as expensive as the *Naive* recency query because the predicate is not selective.

Finally we present the false positive rates of the methods for the test queries. For all the queries used in the experiment, the false positive rates of the *Focused* method are 0. For the *Naive* method, for clarity we compute the false positive rates for one configuration where there are 100000 data sources and assume that the `Routing` table maps the set of machines specified in the query predicates onto itself. The formulas can be verified against the the queries by applying the definition of the false positive rate. Take $Q1$ for example, the rationale behind the expression is that the *Naive* method returns all data sources while there are only 6 of them as specified in the query are relevant.

$Q1$: $fpr = \frac{(100000-6)}{6} = 16665$
$Q2$: $fpr = \frac{6}{(100000-6)} = 0.00006$
$Q3$: $fpr = \frac{(100000-6)}{6} = 16665$
$Q4$: $fpr = \frac{6}{(100000-6)} = 0.00006$

Overall we observe that the overhead of both methods decreases in general as the data ratio increases. When the number of data sources is large and a user query is highly selective on data sources, the *Focused* method is much less expensive than the *Naive* method. Furthermore, the *Focused* method always achieves lower false positive rates than the *Naive* method along with the performance gains.

6. CONCLUSION

In this paper we have argued that when a DBMS is used as a centralized repository to store the state of a distributed, asynchronous job scheduling and execution system, it is more useful and practical to report the recency and consistency of user queries than to enforce traditional notions of consistency. It turns out that if one wants to report recency information for a large system and only wishes to tell the user about data sources that are "relevant" to the user's query, some non-trivial issues arise with defining "relevance." Our solution is to say that a data source is relevant if an update from that data source could possibly change the answer to the user's query.

With this definition of relevance we have developed algorithms for determining relevant data sources, and showed by a theoretical analysis that our techniques will find the minimum except in some extreme cases. We have implemented a working prototype in the Postgres DBMS to explore some practical aspects of doing recency and consistency reporting. Experiments with our implementation show that our techniques incur less overhead in most cases and scale much better than the naive method, in addition to the far better precision achieved with our method for the set of relevant data sources.

Our techniques for recency and consistency reporting are not limited to monitoring grid systems. We think that reporting recency and consistency, rather than enforcing it, will be a viable solution for centralized monitoring and logging of any system comprising a large number of autonomous sources for which it is impractical to obtain and store synchronous global snapshots.

7. ACKNOWLEDGEMENTS

We would like to thank our CondordB group including David J. DeWitt, Ameet Kini, Erik Paulson, Christine Reilly, Eric Robinson, Srinath Shankar, Lakshmikant Shrinivas and Adwait Tumbde who developed the monitoring system for Condor. We are also grateful to the anonymous reviewers for helpful feedback on various drafts of this paper.

8. REFERENCES

[1] R. Alonso, D. Barbará, H. Garcia-Molina, and S. Abad. Quasi-copies: Efficient data sharing for information retrieval systems. In *EDBT*, pages 443–468, 1988.

[2] L. Bright and L. Raschid. Using latency-recency profiles for data delivery on the web. In *VLDB*, pages 550–561, 2002.

[3] D. Byrkit. *Statistics Today*. The Benjamin/Cummings Publishing Company, Inc, 1987.

[4] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB*, pages 471–480, 2001.

[5] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *VLDB*, pages 228–237, 1986.

[6] I. T. Foster and C. Kesselman. Computational grids. In *VECPAR*, pages 3–37, 2000.

[7] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Trans. Database Syst.*, 7(2):209–234, 1982.

[8] H. Guo, P.-Å. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed currency and consistency: How to say "good enough" in SQL. In *SIGMOD Conference*, pages 815–826, 2004.

[9] J. Huang, A. Kini, C. Reilly, E. Robinson, S. Shankar, L. Shrinivas, D. J. DeWitt, and J. Naughton. An overview of quill++: A passive operational data logging system for condor. Technical report, University of Wisconsin at Madison, 2006. https://www.cs.wisc.edu/condordb/overview.pdf.

[10] IBM. *DB2 UDB for z/OS Version 8 Performance Topics*, 2005.

[11] A. Labrinidis and N. Roussopoulos. Balancing performance and data freshness in web database servers. In *VLDB*, pages 393–404, 2003.

[12] R. Lenz. Adaptive distributed data management with weak consistent replicated data. In *SAC*, pages 178–185, 1996.

[13] R. Munz, H.-J. Schneider, and F. Steyer. Application of sub-predicate tests in database systems. In A. L. Furtado and H. L. Morgan, editors, *VLDB*, pages 426–435, 1979.

[14] Oracle Corporation. *Oracle Database Concepts, 10g Release 1*, 2003.

[15] D. J. Rosenkrantz and H. B. Hunt. Processing conjunctive predicates and queries. In *VLDB*, pages 64–72, 1980.

[16] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *ICDE*, pages 512–520, 1990.

[17] University of Wisconsin at Madison. *Condor Version 6.7.17 Manual*, 2002.