

# 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

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

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

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

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 administrators 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

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 of 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  $\langle c_1, c_2, \dots, c_k, c_s \rangle$  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  $\langle v_1, v_2, \dots, v_k, s \rangle$ . The corresponding domains for the columns are denoted  $D_1, D_2, \dots, D_k, D_s$ . 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  $\langle c_1^i, c_2^i, \dots, c_k^i, c_s^i \rangle$  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  $\langle v_1^i, v_2^i, \dots, v_k^i, s^i \rangle$ . The corresponding domains for the columns of  $R_i$  are denoted as  $D_1^i, D_2^i, \dots, D_k^i, D_s^i$ . 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  $\exists v_1 \in D_1, v_2 \in D_2, \dots, v_k \in D_k$  s.t. the tuple  $\langle v_1, v_2, \dots, v_k, s \rangle$  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  $R_1, R_2, \dots, R_n$ , we say that a data source  $s \in D_s$  is **relevant** for  $Q$  if  $\exists i (1 \leq i \leq n), \exists v_1^i \in D_1^i, v_2^i \in D_2^i, \dots, v_k^i \in D_k^i$ , and for  $\forall j (j \neq i, 1 \leq j \leq n), \exists$  a tuple  $\langle v_1^j, v_2^j, \dots, v_k^j, s^j \rangle \in R_j$ , s.t. the tuple  $\langle v_1^i, v_2^i, \dots, v_k^i, s \rangle$  for  $R_i, \langle v_1^j, v_2^j, \dots, v_k^j, s^j \rangle$  for each  $R_j (j \neq i, 1 \leq j \leq n)$  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.

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

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  $R_1, R_2, \dots, R_n$ . We denote the result of  $Q$  as  $Q(R_1, R_2, \dots, R_n)$ . If  $s \notin S(Q)$ , then  $\forall i (1 \leq i \leq n), \forall v_1^i \in D_1^i, \forall v_2^i \in D_2^i, \dots, \forall v_k^i \in D_k^i$ , with  $t_i$  denoting the tuple  $\langle v_1^i, v_2^i, \dots, v_k^i, s \rangle$ , the following holds:*

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

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  $Q(R_1, \dots, R_i \cup \{t_i\}, \dots, R_n) = Q(R_1, \dots, R_i, \dots, R_n) \cup Q(R_1, \dots, \{t_i\}, \dots, R_n)$ . With this we only need to prove that  $Q(R_1, \dots, \{t_i\}, \dots, R_n) = \emptyset$ . Assuming  $Q(R_1, \dots, \{t_i\}, \dots, R_n) \neq \emptyset$ , then for  $\forall j (j \neq i, 1 \leq j \leq n), \exists$  a tuple  $\langle v_1^j, v_2^j, \dots, v_k^j, s^j \rangle \in R_j$ , s.t. the tuple  $t_i$  for  $R_i, \langle v_1^j, v_2^j, \dots, v_k^j, s^j \rangle$  for each  $R_j (j \neq i, 1 \leq j \leq n)$  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  $s \notin S(Q)$ .  $\square$

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  $S(Q)$ .  $\square$

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:  $P_1 \vee P_2 \vee \dots \vee P_k$ , 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:  $P_1 \vee P_2 \vee \dots \vee P_k$  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  $P_2, \dots$   $Q^k$  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  $P_s$ .













