K-Relevance: A Spectrum of Relevance for Data Sources Impacting a Query *

Jiansheng Huang and Jeffrey F. Naughton
Dept. of Computer Science
University of Wisconsin at Madison
1210 W. Dayton St.
Madison, WI 53706, USA
jhuang@cs.wisc.edu, naughton@cs.wisc.edu

ABSTRACT

Applications ranging from grid management to sensor nets to web-based information integration and extraction can be viewed as receiving data from some number of autonomous remote data sources and then answering queries over this collected data. In such environments it is helpful to inform users which data sources are "relevant" to their query results. It is not immediately obvious what "relevant" should mean in this context, as different users will have different requirements. In this paper, rather than proposing a single definition of relevance, we propose a spectrum of definitions, which we term "k-relevance," for $k \geq 0$. We give algorithms for identifying k-relevant data sources for relational queries and explore their efficiency both analytically and experimentally. Finally, we explore the impact of integrity constraints (including dependencies) and materialized views on the problem of computing and maintaining relevant data sources.

Categories and Subject Descriptors

H.1 [**Models and Principles**]: Miscellaneous

General Terms

Theory, Algorithms

*Funded by National Science Foundation Award SCI-0515491

SIGMOD'07, June 11–14, 2007, Beijing, China.

Keywords

k-relevance, k-relevant, relevant sources, lineage relevance, update relevance, materialized view

1. INTRODUCTION

Many modern data management scenarios can be viewed as a set of remote sources that periodically update a centralized database. Examples include sensor data management, distributed system monitoring, and even web-based data integration and extraction (here the remote "sources" are web pages, and the "updates" are the results of running extraction programs that add tuples corresponding to the extracted information to the centralized database.) In such scenarios, the central database is almost always out of date; to make things worse, in some of the scenarios (for example, the extraction scenario) the data is extracted by "fuzzy" routines that may not do a perfectly accurate job, which can frequently cause users to want to do some investigating to understand their results. Interpreting or debugging the results of queries in such systems can be difficult, especially if there are a lot of sources.

Often the work of interpreting or debugging query results requires that the user identify a set of sources to investigate in order to figure out what is going on or to request the most recent data from that source. For example, in an information extraction system, if a user finds that some tuple in the result to her query is surprising, she will likely start her investigation by trying to figure out which source or sources are responsible for the result, in order to see why they might have contributed to the surprising answer. Or, if a user suspects that some result that "should" be in her answer is not there, she needs to find out which sources could have contributed, to see if perhaps they have not updated yet and the problem could be solved by processing an update from that site. In another domain, which we explored in [13], suppose in a distributed job execution system one node says a job has been sent out for execution, but that job is not listed in the database as currently running. The user may wish to find out which nodes could have pending updates that might show the job running.

In all of these examples, the user's task is greatly simplified if the system can restrict the set of nodes she must consider — that is, the system can help the user understand her query results by reporting which data sources were relevant to her query. While this is straightforward, lurking in this simple statement is the undefined phrase "relevant to a query." In this paper we propose a precise definition and

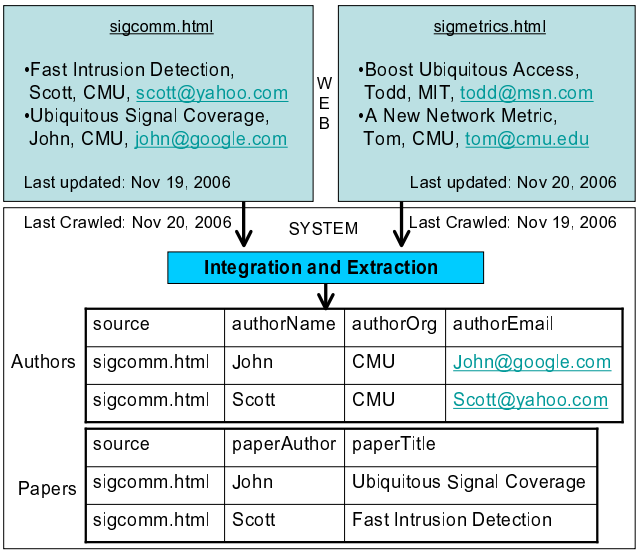


Figure 1: An example web integration and extraction system and sample tables

explore how to compute relevant sources using this definition.

In prior work [13], we defined a data source to be relevant to a query if and only if a single update from that data source could change the result of the query. Although possibly useful in some scenarios, the definition is too narrow to be useful in others. In particular, in some cases a user may just want to know which data sources participated in the derivation of an existing result tuple; telling her which sources "could have" impacted the result may be confusing. On the other hand, in other cases the user may be expecting results that could only arise if there were multiple updates to the database; this could happen if the result will change when one source makes multiple updates, or when several sources each make a single update, or both. In this case sources that are relevant in the users opinion will be deemed irrelevant by the definition from [13]. We illustrate both problems with the following example.

EXAMPLE 1. Figure 1 shows a tiny system that extracts author and paper information from two conference web sites in the networking field and stores it in the two relations: Authors and Papers. The two sources are the web pages sigcomm.html and sigmetrics.html. Note that the "source" column in the tables is system-maintained and that only tuples resulting from extractions from the sigcomm.html page will yield tuples with sigcomm.html in the source field, while only updates from the sigmetrics.html page will yield tuples with sigmetrics.html in the source field. Note also that the extracted database is actually out of date, because the sigmetrics.html page has been updated since it was last extracted.

If a user asks "what are the papers written by authors from MIT with the word 'Ubiquitous' in the titles?," the query will return an empty result. By the definition in [13], the only relevant data source for the query is sigcomm.html. The source sigcomm.html is deemed relevant by that definition because it is possible that an update from this source could change the result of the query — for example, if an update from sigcomm.html changes John's affiliation to MIT, the query result will indeed change to return the paper titled "Ubiquitous Signal Coverage." However, sigmetrics.html is not deemed relevant because no single update from that source to either the Authors or Papers tables in isolation can change the result of the query.

Unfortunately this is misleading for this user, as the sigmetrics.html really is relevant to the query — if the newly updated version of the web page is crawled and extracted, the resulting pair of updates, one each to the authors and papers tables, will change the result, and the paper by Todd from MIT will appear in the answer.

As an example in the other direction, suppose the user now asks the query "Show the names of all authors from CMU." The result will be John and Scott. Furthermore, suppose that the user is surprised at this result because she knows Scott is actually an MIT student. Using the definition from [13], the relevant sources for this query will be both sigcomm.html and sigmetrics.html, since a single update from either could change the result of the query. Again, this is perhaps unfortunate, because it is the (presumably) faulty data at sigcomm.html that is the problem, and the inclusion of sigmetrics.html is at best a distraction and at worst misleading; this user really just wants to know that the data in her answer came from sigcomm.html.

With only two web page sources to consult, telling the user which sources is relevant is perhaps not very important. However, in an example in which there are thousands of sources, properly narrowing the set of "relevant" sources will be the key to the user figuring out what is going on.

It is of course possible to generalize this example to create scenarios in which updates to three relations (possibly from different sites) are required, or four, or any number. In some cases, users may ask "tell me all sources that could impact this query result, no matter how many updates are required." Since it is impossible to identify in advance a single definition of relevance that distinguishes all these different cases and covers them all, in this paper we propose a spectrum of definitions, which we will call "k-relevance", for $k \geq 0$. Intuitively a data source is "k-relevant" if there exist potential updates to k relations, with at least one update from the source in question, that will cause the result of a query to change. The value of k can be interpreted as giving some measure of how closely relevant a source is to a given query.

For the special case of $k = 0$, a data source is "0-relevant" if an existing tuple from that source participates in the derivation of the result for a query. Our solution encompasses the desired notions of relevance in the preceding example: For the first query (all papers written by MIT authors with "Ubiquitous" in the title), sigmetrics.html is 2-relevant, sigcomm.html is 1-relevant, and none is 0-relevant. For the second query (all CMU authors), sigcomm.html is 0-relevant while sigmetrics.thml is 1-relevant.

The set of "k-relevant" sources for a query is a function of the given query and the database instance at the time the query is submitted. Part of the contribution of this paper is algorithms to compute these functions. Another contribution of our paper is to incorporate dependencies and domain constraints and use them to reason about and restrict the set of all possible "k-relevant" sources. (As we shall see, such constraints on the updates sources can be the key to limiting the set of relevant sources.)

Also, we consider the use of materialized views to speed up user queries. The naive way to compute sets of relevant sources for a query is to compute them afresh whenever a query arrives. If the query is common, it may be more efficient to materialize the set of sources relevant to the query. In effect, this treats the query as a view, which can either be materialized or not (if it is materialized, then of course the query itself can be answered from the view; if not, the view is just a mechanism by which to "memo" the set of relevant sources for the query.) Accordingly, we investigate materializing the set of sources relevant to a view, and incrementally maintaining this set. We provide such incremental algorithms by utilizing previous techniques developed for materialized view maintenance. In addition to using materialized view maintenance techniques to incrementally maintain sets of relevant data sources, we explore a dual opportunity: by detecting that updates are from data sources irrelevant to a given materialized view, we can avoid materialized view recomputation, thus improving the efficiency of the maintenance of the view itself.

An interesting question that arises is how users might make use of this notion when interpreting or refreshing their query results — how do they know what kind of relevance they want? The question is largely dependent on the context of a problem a user is dealing with. If a user is debugging a problem with incorrect result and suspects that there are missing updates to at most k relations mentioned in the query, then only k-relevant sources need be investigated and possibly requested for new updates. The choice of k can be made more mechanical in a framework where query results are materialized and maintained incrementally when updates come in from sources. If an incoming update makes changes to k relations mentioned in a query, but the update source is not k-relevant via any of the k relations involved, then applying our theory we can skip the maintenance of the query result for the update even though it makes changes to relations mentioned in the query. The efficiency of such a system is likely to improve because unnecessary computation for view maintenance (detected with our k-relevance) is avoided, while the cost of maintaining k-relevance can be amortized over all database updates.

Finally, we have evaluated the performance of the algorithms for determining k-relevant data sources. Our results show that efficient algorithms exist for small k (k near zero) and for large k (k near $k_{\max}$, the number beyond which k-relevance does not change), although for intermediate values the computation can be expensive.

2. RELATED WORK

Obviously "provenance" or "lineage" is related to our notion of relevance; more precisely, 0-relevance by our definition is closely related to the provenance or lineage discussed in [2, 4, 5, 8, 9]. (It is not identical because we are recording sources that participated in an answer, while other kinds of provenance record more detailed information, for example, individual tuples that participated in the derivation of an answer.) In [10], the authors make use of provenance to help users and system builders debug an information extraction system. Lineage can help users reason about sources that contribute to a query result, but does not directly help users with problems that arise from sources that did not contribute to the query result.

There is little other work that directly focuses on notions related to our concept of "relevance." However, other work has incorporated notions of relevance. For example, in database integration, [15] implemented a system that provides uniform access to a heterogeneous collection of more than 100 information sources. It contains declarative descriptions of the contents and capabilities of the information sources. The system uses the source descriptions to prune the set of information sources irrelevant for a given query and generate executable query plans. The paper does not give an explicit definition for "relevance", but assumes that a data source is implicitly "relevant" if there exists a potential tuple from it that will join with potential tuples from other sources, which is the same as our definition for ∞-relevance. The focus of their work is to eliminate irrelevant sources only in the interest of performance.

In a more tangentially related paper, [7] applies a notion of relevance in topic-specific focused crawling to reduce the number of irrelevant pages that are fetched and discarded. Here the definition of relevance is quite different from ours, as it is a semantic notion that is a probabilistic measure of the distance between a topic and a target page.

3. DEFINING K-RELEVANCE

We assume that each relation which can be updated by any sources contains a data source column that identifies the source where each tuple is from. Notice that our assumption implies that each tuple in such a relation can be updated by one and only one source, therefore two sources will not contradict each other with their updates. We also allow relations that are static, i.e., never affected by updates from the data sources in question. Notice that in reality these relations may not be fixed, because they may be updated through other means (for example, manually by an administrator). Therefore, for the purposes of this paper, by "static relation" we mean a relation that is not affected by data updates directly from the remote sources.

Within the scope of this paper, we assume that a query Q is a single Projection-Selection-Join relational expression with conjunctive predicates that mentions R_1, R_2, ..., R_m,..., R_n, where R_i $(1 \leq i \leq m)$ are updated and R_j $(m < j \leq n)$ are static. Given a relation R_i $(1 \leq i \leq m)$, its data source domain is denoted as D_s. This paper will frequently refer to a "potential tuple" or "real tuple" for a relation. By a "potential" tuple for a relation, we mean a tuple that could be inserted into the relation — that is, it satisfies any constraints on the relation (including domain constraints.) By a "real" tuple we mean one that is found in the current instance of the relation. Lastly, we assume that a list of all data sources known to the database is kept in a static table called H. We use c_s to refer to the data source column of each table which has it.

We begin with a clarification of two kinds of relevance that have fundamentally different semantics. In one kind, a data source is relevant because it has contributed to the result of a query. In the other kind, a data source may not contribute to the result of a query, but is relevant because future updates from that source could cause it to contribute to the result if the query is re-run. We call the first kind of relevance "lineage relevance" and the second kind of relevance "update relevance". More specifically, if a data source can contribute updates to change the result of a query along with updates that can affect at most k relations, we call it "k-relation update relevant". "Lineage relevance" is in fact equivalent

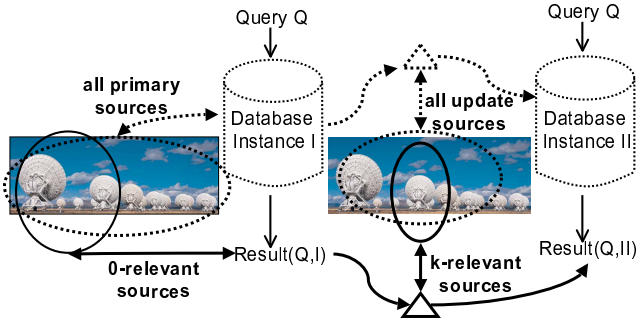


Figure 2: 0-relevant sources contribute lineage tuples to a query result, k-relevant sources contribute updates to changing the query result when updates affect at most k relations

to "0-relation update relevance."

Figure 2 illustrates the two kinds of relevant sources in the flow of database query and updates, where all update sources stand for sources that have contributed updates to the database, while k-relevant sources stand for sources that are k-relevant for the query assuming the updates only affect k relations mentioned in the query. The set of k-relevant sources could be a much smaller subset of the set of all update sources and only updates from k-relevant sources can really contribute to change the query result. In the following, we formally define these concepts, with **k-relevance** standing for "k-relation update relevance" and **0-relevance** standing for "lineage relevance."

3.1 K-Relevance

DEFINITION 1. A data source $s \in D_s$ is **k-relevant** for Q **via** R_i if there exists at most k updated relations including R_i, such that there is a potential tuple from s for R_i if $k > 0$, a potential tuple from any source for the other $k-1$ chosen relations if $k > 1$, and a real tuple for each of the remaining relations such that they join to satisfy Q. A data source $s \in D_s$ is **k-relevant** for Q if there exists R_i such that s is **k-relevant** for Q **via** R_i.

The intuition of this definition is that if there are potential tuples for up to k updated relations that join with real tuples in the remaining relations to satisfy a query, then the data sources that the potential tuples come from are k-relevant. For a query referencing m updated relations, by the definition, k-relevance is equivalent to m-relevance when $k > m$. Obviously any finite m is less than ∞, therefore ∞-relevance is equivalent to m-relevance.

A subtle point in the definition is that when a relation is mentioned multiple times in a query, each mention of it is counted as a different relation for the deciding of k-relevance of a source. For example, assume there are two additional sources now: mobihoc.html and ipsn.html, neither of which has data extracted to the database instance in Example 1. Then the query "Tell me all authors who have published in both MobiHoc and IPSN" can be answered by doing a self join of the Authors table (this answer will be empty for the sample data set.) Therefore, the Authors table appears twice in the query. Because we treat each mention of Authors as a different relation, mobihoc.html is 2-relevant, which matches our intuition because two updates (one from mobihoc.html and the other from ipsn.html), on to each mention of the Authors table in the query, can make a change in the query result.

Another point is that when $k = 0$, a data source is 0-relevant via a relation if there exists a real tuple in the relation from the source that joins with real tuples of other relations to satisfy the query. Therefore a 0-relevant data source participates in the derivation of query answers, which matches our intuitive notion of "lineage relevance."

Given a database with instance I, we denote the set of k-relevant data sources for a query Q via R_i as $S^k_{R_i}(Q, I)$ and the set of k-relevant data sources for a query Q via any relation as $S^k_{all}(Q, I) = \bigcup S^k_{R_i}(Q, I)$. We abbreviate $S^k_{R_i}(Q, I)$ as $S^k_{R_i}$ when Q and I are clear from the context.

EXAMPLE 2. Consider the schema and dataset in Example 1 and the question: "What papers have been published by authors from MIT?" The question can be answered with the following SQL query. The query will return an empty result, therefore it is obvious that $S^0_{Authors} = \varnothing, S^0_{Papers} = \varnothing, S^0_{all} = \varnothing$.

```
SELECT Papers.paperTitle FROM Authors, Papers
WHERE Authors.authorOrg='MIT'
AND Authors.source = Papers.source
AND Authors.authorName = Papers.paperAuthor;
```

Now consider 1-relevant sources. It is possible that sigcomm.html might change some existing author's affiliation from "CMU" to "MIT". Therefore, Authors has a potential tuple from sigcomm.html that could join with a real tuple in Papers to satisfy the query. This means sigcomm.html is 1-relevant via the Authors table. Because all existing authors are from "CMU", no potential tuples of Papers from sigcomm.html will be able to join with a real tuple in Authors to satisfy the query. Therefore sigcomm.html is not 1-relevant via the Papers table. For sigmetrics.html, because neither Authors nor Papers has any data from it, no potential tuples from it to either Authors or Papers alone will be able to join with any real tuple of the other table to satisfy query. Therefore sigmetrics.html is not 1-relevant via any of the tables. To sum up, the 1-relevant sources are the following (Notice that from here on we use s_1 and s_2 as identifiers for sigcomm.html and sigmetrics.html for brevity): $S^1_{Authors} = \{s_1\}, S^1_{Papers} = \varnothing, S^1_{all} = \{s_1\}$.

Now let us consider 2-relevant sources. The source sigmetrics.html can contribute a paper written by someone from "MIT" into the database and such a paper will satisfy the query. Therefore there are potential tuples for Authors and Papers from sigmetrics.html that will join to satisfy the query. This means sigmetrics.html is 2-relevant via both tables. Similarly sigcomm.html can also change the title of a paper and its author to be from "MIT", which means there are also potential tuples from sigcomm.html for both tables which will satisfy the query. Thus sigcomm.html is also 2-relevant via both tables. The summarized result is: $S^2_{Authors} = \{s_1, s_2\}, S^2_{Papers} = \{s_1, s_2\}, S^2_{all} = \{s_1, s_2\}$.

We only need to compute up to 2-relevant sources for the example since there are only two relations mentioned in the query. The sources not in the set of $S^2_{all}(Q, I)$ are always irrelevant to the query and can be ignored for monitoring, debugging or result maintenance purposes because they will never contribute any data to satisfy the query. As an exam-

ple, if another user asks the question: "What papers from the source sigcomm.html have been published by authors from MIT?", then sigmetrics.html is not 2-relevant.

3.2 K-Relevance and Change

We now look into how the type of relevance for data sources updating the tables in the database can be used to determine whether the sources are able to impact a query result and a query's relevant sources. (It is possible that an update, in addition to changing the result of a query, can also change the set of relevant sources for a query.) We start with a result stating that no updates from a data source that is not ∞-relevant will ever impact the result of the query. In other words, a data source that is not ∞-relevant is always irrelevant to the query. The proof is clear and omitted. In the last example of the preceding section, the source sigmetrics.html was not ∞-relevant.

COROLLARY 3.1. *If a source $s \in D_s$ is not ∞-**relevant** for a query Q via a relation R_i, then no updates from the data source to the relation R_i will change the result of Q in any database state.*

If a data source s is not k-relevant via a relation R, any updates from s to R will not change the query result unless more than k other relation updates occur. Notice that the subtlety here is that these updates to the other at most $(k-1)$ relations may in fact change the query result, but the updates to the concerned relation from the non k-relevant data source will not make any difference in the result. Furthermore, this does not mean that the updates from the data source can be ignored forever. When the number of relations being updated exceeds k, these updates may be able to change the query result if the data source happens to be $(k+1)$-relevant. The following results formalize the discussion. The proofs are omitted.

LEMMA 3.2. $S^{k-1}_{R_i}(Q,I) \subseteq S^k_{R_i}(Q,I)$, $k \geq 1$.

COROLLARY 3.3. *For $k \geq 1$, if $s \in S^k_{R_i}(Q,I) - S^{k-1}_{R_i}(Q,I)$, then any updates from s to R_i will not contribute to change the query result until at least $k-1$ other tables are also updated.*

Corollary 3.3 means that if updates are made only to one relation (e.g., R_i), only updates from sources in $S^1_{R_i}(Q,I)$ can possibly change the query result. If updates are made to only two relations (e.g., R_i, R_j), we only need to watch sources in $S^2_{R_i}(Q,I)$ or $S^2_{R_j}(Q,I)$ for updates that are able to make an impact in changing the query result, and so on.

Next we present some results that are related to the potential changes in the "relevant" sources themselves. The proofs for the first two corollaries are omitted because they easily follow from Definition 1.

COROLLARY 3.4. *The sources in $S^\infty_{R_i}(Q,I)$ never change as a result of database updates.*

COROLLARY 3.5. *The sources in $S^k_{R_i}(Q,I)$, $k > 0$, never change as a result of database updates to R_i alone.*

THEOREM 3.6. *If an update is made to a relation R_j and the source for the update is not in $S^\infty_{R_j}(Q,I)$, then $S^k_{R_i}(Q,I)$, for any $k \geq 0$, is unchanged.*

PROOF. We assume $k > 0$ in the proof, as $k = 0$ can be handled similarly. Assume $s \notin S^k_{R_i}(Q,I)$, and after the update to the relation R_j from a source $s_1 \notin S^\infty_{R_j}(Q,I)$, $s \in S^k_{R_i}(Q,I')$ where I' is the new database instance after the update. By the definition, with respect to the instance I there exists a potential tuple for R_i from s, a potential tuple (guaranteed by the update) for R_j from the data source s_1, potential tuples for $k-1$ other relations and a real tuple for each of the remaining relations that satisfy Q. This also means $s_1 \in S^{k+1}_{R_j}(Q,I)$. By Lemma 3.2 $S^{k+1}_{R_j}(Q,I) \subseteq S^\infty_{R_j}(Q,I)$, therefore $s_1 \in S^\infty_{R_j}(Q,I)$. This is a contradiction. Similarly a contradiction can be reached by assuming $s \in S^k_{R_i}(Q,I)$ before the update and $s \notin S^k_{R_i}(Q,I')$ after the update. □

4. COMPUTING K-RELEVANCE

In this section we present algorithms to compute $S^k_{R_i}(Q,I)$ $0 \leq i,k \leq m$, given a database instance I and a query Q, where m is the number of updated relations mentioned in the query. A subtle technicality arises for the computation of $S^k_{R_i}(Q,I)$ if $k > 0$. If a query contains predicates that are not satisfiable at all by potential tuples of mentioned relations, no data source can possibly be k-relevant for the query (because the query's result will be empty for *any* database instance.) We regard this as an unlikely occurrence, hence in our algorithms do not check for query satisfiability (there is ample precedent for this — how many query optimizers check for general satisfiability before executing a query?)

This means that our algorithms are technically only exact with the proviso that the queries themselves do not contain unsatisfiable sets of predicates (such as the predicate $P(x) : x = 1 \wedge x = 2$.) If there are unsatisfiable predicates, our algorithms could deem a source relevant when it really is not, although they will never deem a source irrelevant if it actually is relevant. If a system really needs to remove this proviso so that our algorithms are exact in the presence of unsatisfiable predicates this can be accomplished performing a satisfiability check, such as the one proposed in [17], before passing a query to our algorithms.

4.1 Computing $S^0_{R_i}(Q,I), 1 \leq i \leq m$

One method for computing $S^0_{R_i}(Q,I)$ can be developed based on the work of lineage tracking in [8], where for a given data item in a materialized warehouse view, they presented methods for identifying the set of source data items that produced the view item. Based on their results, they implemented a lineage tracing package in the WHIPS data warehousing system prototype. Their lineage queries return tuples, not data sources; to convert their techniques to return data sources, one would have to add a data source column for each mentioned table. The data source column for each R_i then could be projected out from the lineage results to get $S^0_{R_i}(Q,I)$.

The overhead of this approach can be high as this lineage tracing itself is not cheap. We propose an alternative algorithm shown in Algorithm 1. The idea is to directly modify a user query to include data source columns for each mentioned table and load the result of the modified query into a temporary table, and then query the set of data sources 0-relevant via each table ($S^0_{R_i}(Q,I)$) from the temporary table. Finally the added data source columns can be filtered out from the temporary table before returning the query

Description: Given a query Q and a database with instance I, evaluate the query and derive its 0-relevant sources
Input: a query Q and a database with instance I
Output: query result and $S^0_{R_i}(Q,I), 1 \leq i \leq m$
ALGORITHM 1 (piggyback 0-relevance)
(1) Adorn Q with the data source column for each relation in the returned column list;
(2) Let Q' be the adorned query;
(3) Evaluate Q' and load the result into a temporary table T;
(4) Project out the data source columns for $R_i, 1 \leq i \leq m$ from T;
(5) Assign the result respectively to $S^0_{R_i}(Q,I), 1 \leq i \leq m$;
(6) Assign the union of all $S^0_{R_i}(Q,I)$ to $S^0_{all}(Q,I)$;
(7) Project out the original column list from T for the query result;

result back to a user. We call this a piggyback algorithm because we piggyback the computation of 0-relevant sources with the evaluation of the original user query. In addition to avoiding the extra query evaluation, there is the potential for some savings as we only retain information on the data sources that contributed to the result, not the specific tuples that contributed to the result.

The cost of this algorithm involves writing the result to a temporary table and scanning it twice: once for retrieving the results for $S^0_{R_i}(Q,I), 1 \leq i \leq m$ and the second time for retrieving the result for the original query. The significance of the cost depends on how large a query result is compared to the input tables that the query is scanning and the complexities of the SQL processing.

4.2 Computing $S^\infty_{R_i}(Q,I), 1 \leq i \leq m$

According to the definition of an ∞-relevant source, the current instances of updated relations in the database will not play a role in computing $S^\infty_{R_i}(Q,I)$. What matters are the predicates in the query and constraints (if any) on the column domains for each mentioned relation.

A pre-processing phase should be done first on the query predicates to discover implicit constraints on the data source column of each relation using the transitivity laws of comparison operators. This technique has been studied previously and is known as constant propagation [14]. Integrity constraints including domain constraints that can be expressed in terms of predicates can be simply appended to a user query before our processing. It is less obvious about how to utilize general dependencies, which are especially useful for restricting the relevant data sources. We will address the issue in Section 4.4.

In the following we use P^i_s to stand for the selection predicates referencing only the data source column of R_i, J^i_s to stand for join predicates referencing only the data source column from R_i, and lastly P^i_o to stand for all the other predicates of Q excluding the ones referencing any columns of

Description: Given a query Q and a database with instance I, compute its ∞-relevant sources for the query
Input: a query Q and a database instance I
Output: $S^\infty_{R_i}(Q,I), 1 \leq i \leq m$
ALGORITHM 2 (∞-relevance)
(1) **foreach** $(1 \leq i \leq m)$
(2) Extract P^i_s from Q;
(3) Replace $R_i.c^i_s$ with $H.c_s$ in P^i_s to get $P^{i'}_s$;
(4) Evaluate $\pi_{c_s}(\sigma_{P^{i'}_s}(H))$;
(5) Assign the result to $S^\infty_{R_i}(Q,I)$ and continue;
(6) Assign the union of all $S^\infty_{R_i}(Q,I)$ to $S^\infty_{all}(Q,I)$;

R_i. We use $P^{i'}_s$ and $J^{i'}_s$ to denote the predicates after $R_i.c_s$ is replaced with $H.c_s$ in P^i_s and J^i_s where H is a static table containing all data sources as introduced in Section 3. Given our assumption that Q is satisfiable and all implicit constraints on $R_i.c_s$ are discovered, $S^\infty_{R_i}(Q,I) = \pi_{c_s}(\sigma_{P^{i'}_s}(H))$. This forms the basis for Algorithm 2 to compute ∞-relevant sources.

The main costs of the algorithm are associated with the query generated against the H. The table is static in the sense that it is not affected by updates from data sources and also tend to be much smaller than the updated tables which store application data. In practice we expect that the cost of the algorithm will be small compared to the cost of evaluating a user query and remain flat as a database scales up to larger sizes when new data stream in from sources.

4.3 Computing $S^k_{R_i}(Q,I)$, $1 \leq k < m, 1 \leq i \leq m$

From the definition of k-relevance, we will show that a straightforward method is available to compute $S^k_{R_i}(Q,I)$ when $1 \leq k < m$. However, as we shall see, the cost of the method grows combinatorially as k approaches the middle of the range $1 \leq k < m$. A closer examination reveals that under some conditions the computation of $S^k_{R_i}(Q,I)$ for many values of k can be avoided.

A similar observation we make here is that some join predicates involving the data source column of a relation R_i and another relation may follow implicitly from predicates in a query. For example, if a query has predicates $R.c_s = S.c_1 \wedge S.c_1 = T.c_1$, then $R.c_s = T.c_1$ is an implicit join predicate by transitivity laws. Such implicit join predicates should be discovered and included in J^i_s for computing relevant data sources; otherwise we could miss some implicit constraints useful for computing relevant sources.

When $k \geq 1$, if a data source s is k-relevant via R_i, there exists a potential tuple from s for R_i, a potential tuple for each of $k-1$ more relations, and an existing tuple for each of the remaining relations such that they join to satisfy the query. Let us assume that the potential tuples are for relations $R_{j_1}, ..., R_{j_{k-1}}$ in addition to R_i and the existing tuples are for $R_{j_k}, ..., R_{j_{m-1}}$. Then s is an element of the following set

$$\pi_{c_s}(\sigma_{P^{i'}_s \wedge J^{i'}_s \wedge P^i_o}(H \times \prod_{k \leq l \leq (m-1)} R_{j_l})) \quad (1)$$

The reverse is also true. If s is an element of the set evalu-

Description: Given a query Q and a database instance I, compute its k-relevant sources ($0 < k < m$)
Input: a query Q and a database with instance I
Output: $S^k_{R_i}(Q,I), 1 \leq i \leq m$
ALGORITHM 3 (k-relevance)

(1) **foreach** $(1 \leq i \leq m)$
(2) Extract P^i_s from Q;
(3) Replace $R_i.c^i_s$ with $H.c_s$ in P^i_s to get $P^{i'}_s$;
(4) **foreach** choice of $R_{j_k}, ..., R_{j_{m-1}}$ out of the updated relations except R_i
(5) Extract J^i_s (joining with the selected relations only) from Q;
(6) Replace $R_i.c^i_s$ with $H.c_s$ in J^i_s to get $J^{i'}_s$;
(7) Extract P^i_o (of the selected relations only) from Q;
(8) Evaluate $\pi_{c_s}(\sigma_{P^{i'}_s \wedge J^{i'}_s \wedge P^i_o}(H \times \prod_{k \leq l \leq (m-1)} R_{j_l}))$;
(9) Union the result to $S^k_{R_i}(Q,I)$ and continue;
(10) Assign the union of all $S^k_{R_i}(Q,I)$ to $S^k_{all}(Q,I)$;

ated by (1), then s satisfies the definition of k-relevance via R_i.

The above discussion means that $S^k_{R_i}(Q,I)$, $k \geq 1$ can be computed by doing the union of all such sets evaluated by (1) over all possible ways to choose $R_{j_k}, ..., R_{j_{m-1}}$. Correspondingly we have Algorithm 3 for computing $S^k_{R_i}(Q,I)$.

EXAMPLE 3. This example illustrates Algorithm 3 and the importance of implicit constraints for computing relevant data sources. Suppose the following additional table is added to Example 1:

```
Committee(source, memberName, memberTitle)
```

Assume each table's data source column has the same domain $\{s_1, s_2, ..., s_{100}\}$. Suppose the *Authors* table has tuples from only the three sources $\{s_1, s_2, s_3\}$, the *Papers* table has tuples only from $\{s_1\}$ and the *Committee* table has tuples only from $\{s_2\}$. A user asks "which papers have authors that are committee members, and what are the affiliations of those authors?" and answers it by joining these tables on the data source column with the following query:

```
SELECT Papers.paperTitle, Papers.paperAuthor,
       Authors.authorOrg, Committee.memberTitle
FROM Authors, Papers, Committee
WHERE Authors.source = Papers.source
AND Papers.source = Committee.source
AND Authors.authorName = Papers.paperAuthor
AND papers.paperAuthor = Committee.memberName;
```

For this query, if we do not derive implicit constraints before applying Algorithm 3, then $S^2_{Authors}(Q,I)$ is equal to the following expression which evaluates to $\{s_1\} \cup \{s_1, s_2, ..., s_{100}\} = \{s_1, s_2, ..., s_{100}\}$.

$$\pi_{c_s}(\sigma_{H.c_s = Papers.source}(H \times Papers)) \cup \pi_{c_s}(H \times Committee)$$

On the other hand if we derive the implicit constraint "`Authors.source = Committee.source`" before applying Algorithm 3, we will have the following expression instead for $S^2_{Authors}(Q,I)$ which now evaluates to $\{s_1\} \cup \{s_2\} = \{s_1, s_2\}$. The answer is much smaller than the previous result. This shows that the implicit constraints on the data source column can be critical in computing relevant data sources.

$$\pi_{c_s}(\sigma_{H.c_s = Papers.source}(H \times Papers)) \cup \pi_{c_s}(\sigma_{H.c_s = Committee.source}(H \times Committee))$$

In Algorithm 3, the number of ways to choose $R_{j_k}, ..., R_{j_{m-1}}$ out of all updated relations except R_i is $\frac{(m-1)!}{(m-k)!(k-1)!}$. The combinatorial number indicates that this approach is expensive for any k values in the middle of the range when m is large. Therefore any alternative that is more efficient to obtain the result for $S^k_{R_i}(Q,I)$ is preferred.

A useful condition for avoiding the expense of Algorithm 3 is when $S^k_{R_i}(Q,I)$ is equal to $S^h_{R_i}(Q,I)$ for some pair of $0 \leq k < h \leq m$. When this happens, there is no need to compute $S^j_{R_i}(Q,I)$ for $k < j < h$ because it has to be the same as $S^h_{R_i}(Q,I)$ given the monotonicity property of $S^k_{R_i}(Q,I)$ stated in Lemma 3.2. Notice that to check this condition one only has to keep cardinalities for $S^k_{R_i}(Q,I)$ and compare them between two sets instead of comparing element by element. Given this alternative and the observation that the above combinatorial number is first increasing and then decreasing after k passes a median number between 1 and m, the best order to compute $S^k_{R_i}(Q,I)$ is to alternate between the two ends and close in, i.e., $0, m, 1, m-1, 2, m-2, ...$ until the list is exhausted.

Lastly, if the previous condition fails to avoid the computation for some $S^k_{R_i}(Q,I)$, $1 \leq k < m$, Theorem 4.1 provides another useful condition that can be checked to see if there is still a possibility to avoid it.

THEOREM 4.1. *If there exists a combination $R_{j_k}, ..., R_{j_{m-1}}$, of which none is joined with R_i through the J^i_s of a query Q, and $\sigma_{P^i_o}(\prod_{k \leq l \leq (m-1)} R_{j_l})$ is not empty, then $S^k_{R_i}(Q,I)$ is equal to $S^\infty_{R_i}(Q,I)$.*

PROOF. Given the combination $R_{j_k}, ..., R_{j_{m-1}}$, the set of k-relevant sources computed through Algorithm 3 includes at least the following:

$$\pi_{c_s}(\sigma_{P^{i'}_s \wedge P^i_o}(H \times \prod_{k \leq l \leq (m-1)} R_{j_l}))$$

Notice that there is no join predicates between H and any of $R_{j_k}, ..., R_{j_{m-1}}$ in the expression because none is joined with R_i through J^i_s. Because $\sigma_{P^i_o}(\prod_{k \leq l \leq (m-1)} R_{j_l})$ is not empty, the above expression is the same as $\pi_{c_s}(\sigma_{P^{i'}_s}(H))$, which is the result for $S^\infty_{R_i}(Q,I)$ as computed by Algorithm 2. □

4.4 Source - Non-Source Dependencies

As we have hinted earlier on, often the implementers of databases that are repositories for distributed sources have information about data values that can come from the individual sources. In our toy example, perhaps we could add a "confName" field and note that sigcomm.html will only generate tuples with "confName" set to "SIGCOMM." Or in a

sensor network application, if each source is a (non-mobile) sensor, an "area of coverage" field might be restricted based upon the identity of the sensor. More generally, we may know that a bookseller web site will only publish advertisements for books. Such dependencies can be the key to narrowing sets of relevant sources for some queries.

The dependencies we support here are not limited to functional dependencies. For example in an e-commerce application site-1 can generate books and videos tuples, site-2 can generate books and cd's tuples. In this paper we propose modeling such dependencies by a separate static lookup table. Notice that the static table modeling a dependency this way does not have to maintain a tuple for every potential "key" value in its domain (which could be infinite), but only values for sources known to the system. Finally, a dependency modeled this way can be enforced on target tables by creating a foreign key on the list of key columns and the data source column which references the primary key on the same set of columns of the static lookup table.

The following theorem takes a materialized dependence table into consideration for computing ∞-relevant sources. To simplify our notation we assume only one column is associated with the data source column in a dependency in the following. We use F_i to stand for a static table modeling a dependency of the table R_i from a regular column (referred to as the "key" column) to the data source column. The "key" column of the dependency has the column name c_f^i in both F_i and R_i. P_f^i is the predicate in the query Q which references only the c_f^i column of R_i. $P_f^{i'}$ is the resulting predicate after the variable $R_i.c_f^i$ is replaced by $F_i.c_f^i$ in predicate P_f^i.

THEOREM 4.2. *If R_i has a dependency from a column c_f^i to its data source column and the dependency is captured in a static table F_i, then*

$$S_{R_i}^{\infty}(Q,I) = (\pi_{c_s}(\sigma_{P_s^{i'}}(H)) \cap \pi_{c_s}(\sigma_{P_f^{i'}}(F_i)))$$

Algorithm 2 can be modified accordingly to incorporate the use of dependencies. To save space, the extension to incorporate dependencies into Algorithm 3 is omitted due to space considerations.

5. MAINTAINING K-RELEVANCE

Materialized view maintenance has been studied widely in the past [1, 3, 6, 11, 12, 16, 18, 19]. In this section we discuss efficient strategies for a related problem — maintaining up-to-date relevant sources for a query in response to database updates. The idea is that for a commonly asked query, it may be better to materialize the set of relevant sources for that query once and maintain it incrementally than to compute it from scratch every time the query is asked. Furthermore, we will show that the relevant sources information we maintain can also be leveraged in reducing the cost of maintaining the query result itself (that is, the materialized view defined by the query.)

The view maintenance algorithms presented in [11] can be used to maintain materialized query results. We can adopt these algorithms to maintain sets of relevant sources. One reasonable algorithm is the counting algorithm [12], which we will use as a basic building block. We describe the main steps in Algorithms 4 and 5 respectively for the maintenance of $S_{R_i}^k(Q,I)$ $(1 \le k < m)$ and $S_{R_i}^0(Q,I)$. Algorithm 5 incorporates the maintenance of 0-relevant sources into the maintenance of query results to further reduce the cost. Notice that Algorithm 5 relies on $S_{R_i}^k(Q,I)$, therefore in practice Algorithm 4 should be run before running Algorithm 5 to guarantee correctness. There is no need to maintain $S_{R_i}^{\infty}(Q,I)$ because it never changes according to Corollary 3.4.

Description: Given a query Q and a database with instance I, maintain the k-relevant sources for the query
Input: materialized results for $S_{R_i}^k(Q,I)$ and updates
Output: new results $S_{R_i}^k(Q,I), 1 \le i \le m, 1 \le k < m$

ALGORITHM 4 (k-relevance maintenance)

```
(1)  foreach (update)
(2)      if (the update is made to relation R_j and the source for
             update is not in S^∞_{R_j}(Q,I))
(3)          ignore the update in this algorithm;
(4)  if (no more updates left)
(5)      no need to do maintenance at this time, return;
(6)  foreach (1 ≤ i ≤ m)
(7)      foreach (update)
(8)          if (update is made to R_i)
(9)              ignore the update for the maintenance of S^k_{R_i}(Q,I);
(10)     if (no more updates left for the maintenance of S^k_{R_i}(Q,I))
(11)         no need to do maintenance for S^k_{R_i}(Q,I) at this time, continue;
(12)     foreach (1 ≤ k < m)
(13)         Use the counting algorithm to compute delta for Q^k_i with the remaining updates;
(14)         Apply the delta to the materialized result of S^k_{R_i}(Q,I);
```

In Algorithm 4, Q_i^k stands for the query that we build in Algorithm 3 for computing each $S_{R_i}^k(Q,I)$. The algorithm's basic idea is to apply Corollary 3.5 and Theorem 3.6 to skip updates that cannot possibly change $S_{R_i}^k(Q,I)$, and then to process surviving updates using the counting algorithm.

In Algorithm 5, Q' is the same as the adorned query produced in Algorithm 1. To simplify the description of the algorithm, we assume that there is a count associated with every data source in $S_{R_i}^0(Q,I)$. If the count is zero for a data source, it means the data source does not belong to $S_{R_i}^0(Q,I)$. For clarity, we also assume that the count and data sources with count zero will be filtered out before the result is presented to a user. Therefore the maintenance algorithms we describe here ignore such issues so that we can focus on the key steps.

This algorithm is different from previous view maintenance algorithms in that it leverages information about the

relevant sources for the query defining the view to ignore updates from sources that cannot affect the materialized view in a maintenance call. Specifically by applying Corollary 3.3, Algorithm 5 ignores updates to a relation R_j if their data sources are not in $S^k_{R_j}(Q, I)$.

Description: Maintenance algorithm for the query result of Q and its 0-relevant sources
Input: materialized results for Q, Q', $S^0_{R_i}(Q, I)$ and updates
Output: new results for Q, Q' and $S^0_{R_i}(Q, I), 1 \leq i \leq m$
ALGORITHM 5 (0-relevance maintenance)

```
(1)   Let k = the number of relations
      all updates are made to;
(2)   foreach (update)
(3)     if (the update is made to re-
        lation R_j and the source for
        update is not in S^k_{R_j}(Q, I))
(4)       ignore the update in this
          algorithm;
(5)   if (no more updates left)
(6)     return;
(7)   Use the counting algorithm to
      compute delta for Q';
(8)   Apply the delta to the material-
      ized result of Q';
(9)   foreach (tuple in the delta)
(10)    foreach (1 ≤ i ≤ m)
(11)      Project out the data
          source column for R_i;
(12)      if (the tuple is to be in-
          serted)
(13)        Add the count in
            the tuple to the
            corresponding tuple
            in S^0_{R_i}(Q, I) and
            S^0_{all}(Q, I) with the
            same data source;
(14)      else if (the tuple is to be
          deleted)
(15)        Subtract the count
            in the tuple from
            the corresponding
            tuple in S^0_{R_i}(Q, I) and
            S^0_{all}(Q, I) with the
            same data source;
(16)    Project out the original col-
        umn list from the tuple and
        apply to the materialized re-
        sult of Q;
```

6. EXPERIMENTAL EVALUATION

The goal of our experiments is 1) to compare the cost of the computation and maintenance of k-relevant sources relative to the cost for the original query; 2) to understand the dependence of this overhead on the number of data sources and the size of data; 3) to gain some insight into how effective our algorithms are in limiting the number of relevant data sources.

Our experimental schema contains the tables listed in the following. Both Authors and Papers have a domain constraint enforcing their sourceIds to have "conf" as a prefix, and Students has a domain constraint enforcing its sourceIds to have "student" as a prefix. The static ConfSourceFD table is an explicit representation of a dependency from (confName, confYear) to sourceId for Authors and Papers. The static StudentSourceFD table models the dependency from (name, org) to sourceId for Students. The static AllSources table contains the ids for all sources. We created B-tree indices for each set of underlined columns.

```
Authors (sourceId, confName, confYear, name,
         org, position, email)
Papers (sourceId, confName, confYear,
        authorName, authorOrg, title)
Students (sourceId, name, org, advsr, prog, yr)
ConfSourceFD (confName, confYear, sourceId)
StudentSourceFD (name, org, sourceId)
AllSources (id, url)
```

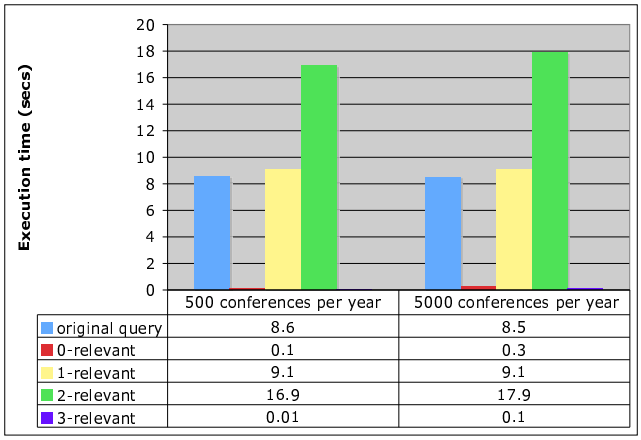
When generating datasets, we assumed each type of conference has been held for 30 years. We also assumed there were 100 author organizations, each of which has 10 students and 5 professors. Three datasets are generated with the following combinations: 1) 500 conferences per year and 200 papers per conference, 2) 5000 conferences per year and 200 papers per conference, 3) 500 conferences per year and 400 papers per conference. Lastly we also simulate missing updates from data sources by allowing a small percentage of conferences or students randomly chosen to not have any data, with the meaning that the data from these sources are missing. All the conferences are independently generated and the authors for each paper are randomly chosen from the pool of students and professors using a uniform distribution.

To evaluate maintenance cost of materialized views, we simulate updates with pairs of delta tables for Authors and Papers. All the data in one pair of delta tables are from 0-relevant sources, which is the worst case scenario for the maintenance of Q and its k-relevant sources. For the other pair of delta tables, all the data are from sources that are not ∞-relevant.

The following experimental query (referred to as Q) finds papers published by students in a list of 10 conferences with year unspecified and for each student reports the program in which he studies, the year of graduation, and his advisor.

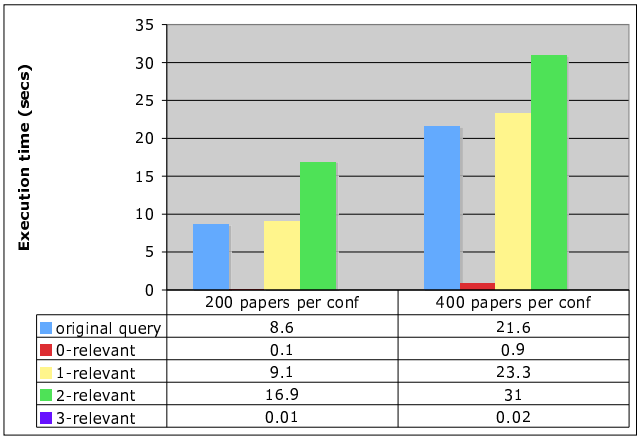
```
SELECT A.name,A.org,P.title,S.prog,S.yr,S.advsr
FROM Authors AS A, Papers AS P, Students AS S
WHERE A.confName IN [list of 10 conf types]
AND A.position = 'student'
AND A.name = P.authorName
AND A.org = P.authorOrg
AND A.confName = P.confName
AND A.confYear = P.confYear
AND A.name = S.name AND A.org = S.org
```

For the algorithms, we only evaluated the costs of queries generated for computing relevant sources, which did not include costs for pre- and post-processing (e.g., SQL parsing, result merging etc.) Our experimental setup used Tao Linux 1.0 on top of a 2.4 GHZ Intel Pentium with 512MB memory



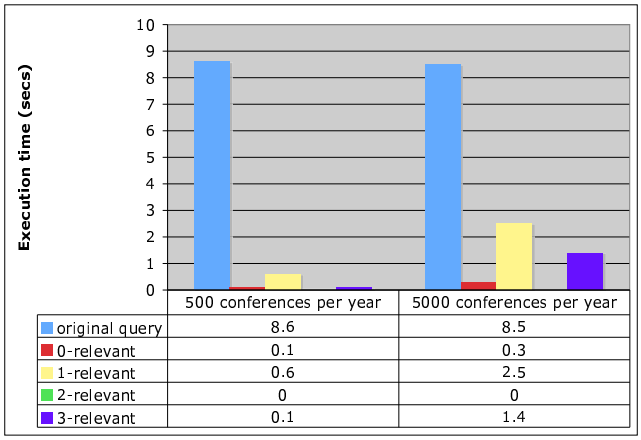
	500 conferences per year	5000 conferences per year
original query	8.6	8.5
0-relevant	0.1	0.3
1-relevant	9.1	9.1
2-relevant	16.9	17.9
3-relevant	0.01	0.1

Figure 3: Comparison of costs for evaluating Q and k-relevant sources for Q via Students with variation on the number of conferences per year.



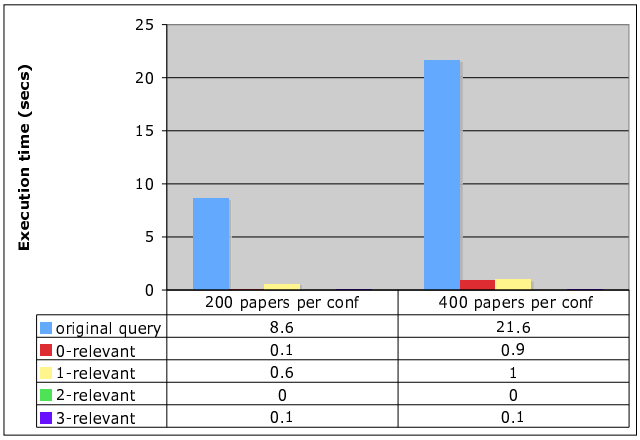
	200 papers per conf	400 papers per conf
original query	8.6	21.6
0-relevant	0.1	0.9
1-relevant	9.1	23.3
2-relevant	16.9	31
3-relevant	0.01	0.02

Figure 5: Comparison of costs for evaluating Q and k-relevant sources for Q via Students with variation on the number of papers per conference



	500 conferences per year	5000 conferences per year
original query	8.6	8.5
0-relevant	0.1	0.3
1-relevant	0.6	2.5
2-relevant	0	0
3-relevant	0.1	1.4

Figure 4: Comparison of costs for evaluating Q and k-relevant sources for Q via Authors with variation on the number of conferences per year.



	200 papers per conf	400 papers per conf
original query	8.6	21.6
0-relevant	0.1	0.9
1-relevant	0.6	1
2-relevant	0	0
3-relevant	0.1	0.1

Figure 6: Comparison of costs for evaluating Q and k-relevant sources for Q via Authors with variation on the number of papers per conference

running the PostgreSQL 8.1.5 database platform. The default settings were used for the shared buffer pool size (8MB) and work memory (1MB). Each query was run 11 times and we report the response time averaged over the last 10 runs.

Figures 3 and 4 compare the costs for evaluating Q and the k-relevant sources for Q via Students and Authors while varying the number of sources. Both figures show that the costs of 0-relevant and ∞-relevant sources are insignificant compared to the cost of the original query. While costs of 1-relevant and 2-relevant sources for Q via Authors are not significant, the cost of sources 1-relevant via Students is approximately as high as the cost of Q, and the cost of sources 2-relevant via Students is twice as high as the cost of Q. This is because the queries generated for computing sources relevant via Students access more large tables than those generated for computing sources relevant via Authors. The results agree well with our analysis in Section 4. The costs of 0-relevant sources are low because this does not require any additional query processing. The costs of ∞-relevant source are also low because they only require access to AllSources and the dependence tables, which are much smaller than the other tables. Notice that the costs for sources 2-relevant via Authors are zeros because there is no need to compute them according to Theorem 4.1.

When the number of conferences per year increases by a factor of 10 from 500 to 5000, it affected 3-relevant sources the most in terms of the cost increase ratio. This is mainly because the queries for evaluating them all need to do a full scan of AllSources to evaluate the domain constraints. While queries for evaluating 1-relevant or 2-relevant sources also contain full scans of AllSources for checking domain constraints, their cost increase ratio are lower because the accesses to Authors or Papers or both in these queries dominate the cost and the cost of index accesses over these tables are not affected by the increase of the number of conferences.

Figures 5 and 6 investigate the cost of computing sources k-relevant via Students and Authors while varying the size of the data. The result shows that the cost overhead ratios for computing k-relevant (except 0-relevant) sources are stable or decreasing as the size of data increases. Specifically, the costs for 1-relevant sources and 2-relevant sources increase proportionally with the cost of Q because they all access Au-

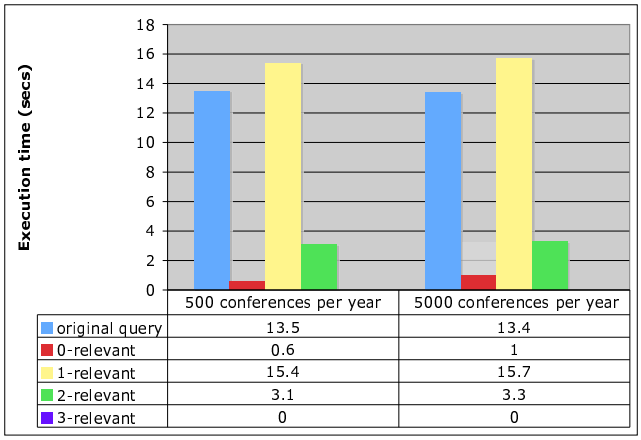


Figure 7: Comparison of costs for maintaining materialized views for Q and k-relevant sources for Q via Students when updates are from 0-relevant sources

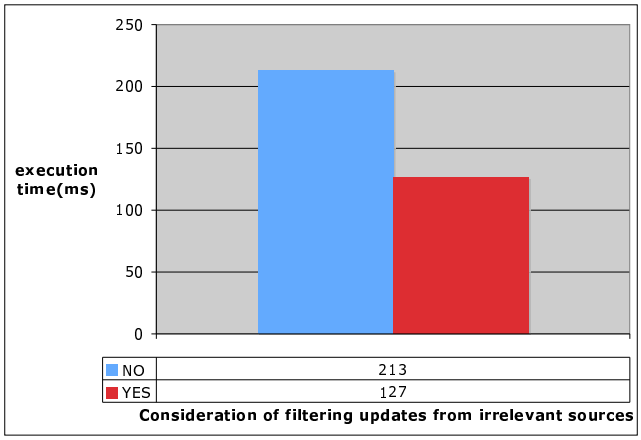


Figure 8: Comparison of costs for maintaining materialized query result with and without filtering of updates from sources not ∞-relevant

thors or Papers or both, which are affected by the increase of papers per conference. The cost for 3-relevant sources is roughly unchanged because their computation only involves access to the AllSources or dependence tables, which are not affected by the papers per conference factor. Finally, for 0-relevant sources, the cost overhead is insignificant.

Figure 7 compares the cost of maintenance for the materialized views of Q and sources k-relevant via Students while varying the number of conferences per year. It shows that the overhead for the maintenance of k-relevant sources with respect to the maintenance of the materialization of Q is at worst similar to the overhead for the computation of k-relevant sources with respect to the computation of Q shown in Figures 3, 4, 5 and 6. Figure 8 shows that the cost for maintaining the materialized query result can be reduced by filtering out updates from irrelevant sources. Notice that the more an updated table is mentioned in a query, the larger the savings will be due to the reduction in the amount of updated data a maintenance query has to access.

Figure 9 gives an example where the cost of computing k-relevant sources is very high. The example query does a

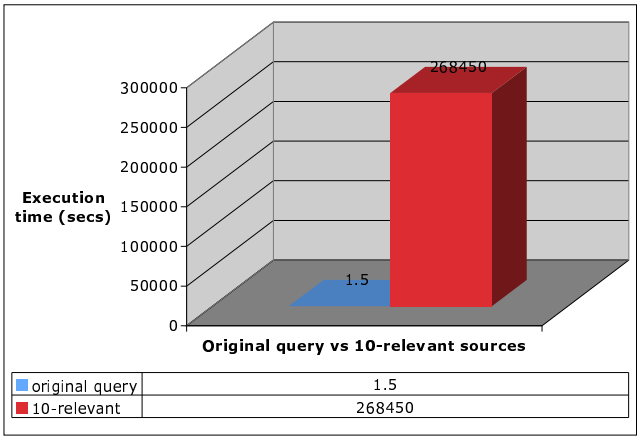


Figure 9: A scenario when k-relevant sources is expensive to compute

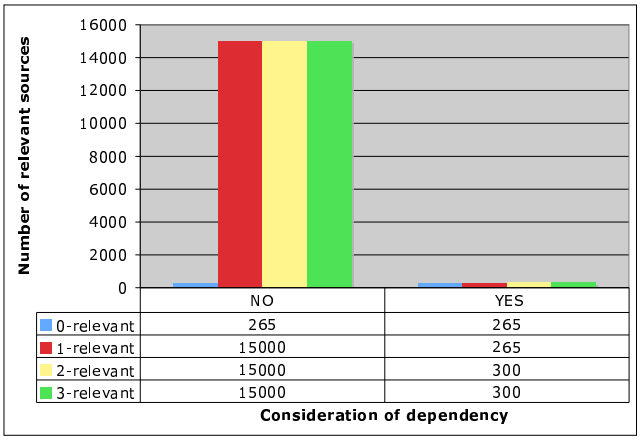


Figure 10: The importance of using dependency in computing k-relevant sources

20-way self join on the Students table to print the names of each student 20 times. The query has a chain join on the name and organization columns of Students. The result shows that it is very expensive to compute 10-relevant sources for such a query. The high cost is largely due to the number of ways to choose 10 relations out of 19 relations for the computation of 10-relevant sources according to Algorithm 3. While our algorithms can be improved to avoid redundant evaluations of the same query generated for this particular example, it provides an idea on how bad the cost of computing k-relevance can be in a complex query with lots of joins on data source columns.

Figure 10 compares the cardinalities of sources k-relevant via Authors computed with and without consideration of dependencies. The experimental query has predicates specified on the columns that determine the data source, but not directly on the data source column (e.g., `A.confName IN [list of 10 conf types]`.) The result indicates that dependencies can dramatically reduce the number of k-relevant sources when $k > 0$.

The experimental results have validated our argument that the costs of 0-relevant and ∞-relevant sources are relatively insignificant compared to the cost of a query itself.

If a query has limited number of join predicates on the data source column or those determining it through dependencies, the cost of computing k-relevant sources is not far from that of computing the query itself in the worst cases. However, if the query has a large number of join predicates on such columns, the cost could be prohibitive, in which case it may be wiser to just use the upper bound provided by the computation of ∞-relevant sources.

The conclusions from our experiments that can be applied to real-world data sets are more qualitative than quantitative - for example, in our experiments, the overhead is low for small k and large k, but more substantial for intermediate values. We expect our experimental results to be a general trend that holds for real-world data sets as well. We evaluated the cost of k-relevance for each k separately to expose the difference in costs associated with each k. If a user is interested in multiple k's, the cost needs to be summed up over these k's.

7. CONCLUSION AND FUTURE WORK

In any data management scenario in which data from remote sources is captured, stored, and queried in a central repository, in their quest for interpreting or debugging their query results, users can benefit from being told which sources were relevant to their query. This is especially true in scenarios where there are lots of sources and parts of the centralized database are likely to be out of date, or the data acquired from the remote sources is potentially imprecise, or both. It turns out that defining what is meant by "relevant" is non-trivial and there exists a rich structure of different types of relevance for different queries and data configurations. In this paper we have proposed a spectrum of definitions for relevance unified under a common notion of k-relevance.

Our notion of k-relevance essentially captures whether updates to a relation from a source are able to change the result of query, along with updates to additional $k-1$ relations. An interesting area for future work is to explore other notions of relevance that may be more intuitive to end users. For example, it may be desirable to tell a user which specific sources or transactions need to update the database before the result of a query could possibly change. We hope that our paper serves as the groundwork that will pave the way for this effort.

8. ACKNOWLEDGMENTS

The authors are grateful to AnHai Doan, Chen Li and three anonymous reviewers for helpful feedback on various drafts of this paper.

9. REFERENCES

[1] D. Agrawal, A. E. Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *SIGMOD Conference*, pages 417–427, 1997.

[2] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, pages 953–964, 2006.

[3] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, pages 61–71, 1986.

[4] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

[5] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[6] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, pages 577–589, 1991.

[7] S. Chakrabarti, K. Punera, and M. Subramanyam. Accelerated focused crawling through online relevance feedback. In *WWW*, pages 148–159, 2002.

[8] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *ICDE*, pages 367–378, 2000.

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

[10] A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen. Community information management. *IEEE Data Eng. Bull.*, 29(1):64–72, 2006.

[11] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[12] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 157–166, New York, NY, USA, 1993. ACM Press.

[13] J. Huang, J. F. Naughton, and M. Livny. Trac: Toward recency and consistency reporting in a database with distributed data sources. In *VLDB*, pages 223–234, 2006.

[14] M. Jarke and J. Koch. Query optimization in database systems. In *ACM Comput. Surv.*, volume 16, pages 111–152, New York, NY, USA, 1984. ACM Press.

[15] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, pages 251–262, 1996.

[16] H. Liefke and S. B. Davidson. View maintenance for hierarchical semistructured data. In *DaWaK*, pages 114–125, 2000.

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

[18] O. Shmueli and A. Itai. Maintenance of views. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 240–255. ACM Press, 1984.

[19] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD Conference*, pages 316–327, 1995.