Recommending Join Queries Via Query Log Analysis

Xiaoyan Yang #1, Cecilia M. Procopiuc *2, Divesh Srivastava *3

National Univ. of Singapore, Republic of Singapore, 117543
1xiaoyan@nus.edu.sg

* AT&T Labs-Research, Florham Park, NJ 07932 USA
2magda@research.att.com
3divesh@research.att.com

***Abstract*— Complex ad hoc join queries over enterprise databases are commonly used by business data analysts to understand and analyze a variety of enterprise-wide processes. However, effectively formulating such queries is a challenging task for human users, especially over databases that have large, heterogeneous schemas. In this paper, we propose a novel approach to automatically create join query recommendations based on input-output specifications (i.e., input tables on which selection conditions are imposed, and output tables whose attribute values must be in the result of the query). The recommended join query graph includes (i) "intermediate" tables, and (ii) join conditions that connect the input and output tables via the intermediate tables. Our method is based on analyzing an existing query log over the enterprise database. Borrowing from program slicing techniques, which extract parts of a program that affect the value of a given variable, we first extract "query slices" from each query in the log. Given a user specification, we then re-combine appropriate slices to create a new join query graph, which connects the sets of input and output tables via the intermediate tables. We propose and study several quality measures to enable choosing a good join query graph among the many possibilities. Each measure expresses an intuitive notion that there should be sufficient evidence in the log to support our recommendation of the join query graph. We conduct an extensive study using the log of an actual enterprise database system to demonstrate the viability of our novel approach for recommending join queries.**

I. Introduction

Enterprise databases typically have hundreds of tables, with a huge number of potential join edges between pairs of tables, making it particularly challenging to understand their structure and to formulate non-trivial join queries. Business data analysts are often faced with exactly this challenging task, since they need to pose complex ad hoc join queries to perform sophisticated analyses of a variety of enterprise-wide processes whose data is buried in these databases. Graphical tools for query formulation are helpful when the analysts already have a comprehensive understanding of the schemas and the tables that need to be joined for the analysis task at hand. When the analyst does not have such a thorough understanding of the database (which is quite common in practice), formulating such complex SQL join queries becomes tedious, especially over databases with large, heterogeneous schemas.

In this paper, we propose a novel approach of using the SQL query log of the enterprise database to automatically create join query recommendations for a user (e.g., data analyst). The user needs to provide only an "input-output" specification, consisting of (i) the input (or *start*) tables on which conditions of the form "$R.f = value$" are present in the SQL WHERE clause, and (ii) the output (or *end*) tables whose attribute values are in the result of the query, i.e., in the SQL SELECT clause. The recommended join query includes (i) intermediate tables that are added to the SQL FROM clause, and (ii) join conditions that are added to the SQL WHERE clause, and connect the input and output tables via the intermediate tables.

When the desired query is a complex ad hoc join query, the SQL query log is unlikely to contain any query whose sets of start and end tables exactly match the user specifications. In such a scenario, our approach analyzes the rich syntactic and semantic structure of join queries in the SQL query log, and combines information from multiple previously posed queries to construct a good join query recommendation for the user.

To construct such a query, we must first understand how different combinations of start and end tables influence which join edges are instantiated in different logged queries. We propose to view each join query in the SQL log as a small program, whose initialization statements correspond to the choice of start tables and, following a flow of dependencies given by the join edges, we arrive at the end tables, from which the result is returned to the user. Drawing this parallel between join queries and programs allows us to use the program-slicing framework defined in the seminal paper by Weiser [9]. A program slice consists of parts of a program that (potentially) affect values computed at some point of interest in the program. New programs can be created by extracting and combining slices from existing programs [5], [8].

In our model, each end table constitutes a point of interest, and we will define a *query slice* as a succession of join edges (i.e., a path) that starts in a start table and ends in an end table in a query in the log. We view a slice in the static sense, i.e., we assume that any column or value can be used in the selection conditions of its start table, without altering the dependency flow. Once such slices are extracted from each join query in the SQL query log, we recombine them to construct new join queries, based on the user's input-output specification.

There are many different ways of choosing which set of slices to recombine to generate a new join query. To evaluate the quality of the generated queries, we assume the existence

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.122

of an *oracle* which can recommend an ad hoc join query for any user specification. Such an oracle has a well defined, but unknown, set of rules, derived from complete knowledge of the schemas and semantics of the underlying databases. Our goal is to emulate the oracle's behavior, without knowing its set of rules, by analyzing the SQL query log which is assumed to be consistent with the oracle. To this end, we propose and study several quality measures to enable choosing a good join query among the many possibilities. Each quality measure expresses an intuitive notion that there should be sufficient evidence in the query log to support our join query recommendation. We propose algorithms for generating complex join queries under each measure, and conduct an experimental study to evaluate the relative merits of each measure.

In summary, we make the following contributions:

1) We propose a novel approach of automatically generating join queries that match an input-output specification. Our approach is based on the analysis of SQL logs, and is inspired by ideas from program slicing.
2) We identify several quality measures to enable choosing a good join query among the many possibilities, and design efficient algorithms for generating complex join queries under each measure we identify.
3) We perform an extensive experimental study, using the query log of an actual enterprise database system, to demonstrate the viability of our novel approach for recommending join queries.

A. Related Work

To the best of our knowledge, there is no prior work on SQL query log mining for recommending join queries.

The idea of allowing a user to specify only a part of the query and have the system recommend a completed query was explored by [6]. Their approach considers object-oriented queries with incomplete, ambiguous path expressions as input, and generates queries with fully specified path expressions that are consistent with the input, by exploiting the semantics of the relationships in the schema.

Data-driven approaches have been proposed for approximating the similarity between two database columns [4]. Such approaches provide only local information about the schema (e.g., join edges between tables). In order to connect information across distant tables, further exploration of the schema graph must be employed, to determine which join paths to instantiate.

In the machine learning community, the problem of discovering frequently instantiated join paths was considered by [10], formulated as the discovery of multi-relational subgroups in the data. The approach is an on-the-fly exploration of the schema, starting with a random sample in a table, and executing a database join for each edge traversal (and additional sampling of results).

A related area is that of keyword search in relational databases (e.g., [2], [7]). A query consists of several keywords, and the answer is computed by joining tuples from multiple tables into tuple trees. Tuples in the leaves must contain at least one query keyword each. The user need not know the join structure of the schema, but the query processor does.

Techniques for analyzing query logs have been intensively studied in the past decade, in conjunction with keyword searches over the web. The analysis of logs can serve a variety of purposes, such as improving URL recommendations for frequently asked queries, building query taxonomies [3], or understanding user behavior (i.e., clicked links) associated with queries [1]. Unlike web logs, queries in database logs have a rich syntactic and semantic structure, which makes them challenging to analyze.

II. Preliminaries

Let $\mathcal{R}$ be the set of relations in a database. We denote by $\mathcal{A} = \{A_1, \ldots, A_k\}$ the set of attributes in the database. Each table $R \in \mathcal{R}$ has fields $R.f_1, R.f_2, \ldots$ such that the domain of each $R.f_i$ is included in some set A_j. The schema graph is defined in the usual manner: there is a node in the graph for each table, and a join edge between tables R_1 and R_2 for each pair of columns $R_1.f_1$ and $R_2.f_2$ that can be joined (thus, there may be multiple edges between the same pair of graph nodes). Self-joins are represented as loops. Each join query Q represents a subgraph in the schema graph, containing all join edges that appear in Q. If Q references views, then they are replaced, in a pre-processing step, by the tables over which the views are defined (this may also introduce additional edges to the subgraph of Q, for views defined via joins). We decompose join query subgraphs into slices, as defined below.

Definition 1:
1) Let Q be a join query. The *start tables* of Q are those on which there are selection conditions in the WHERE clause of Q (i.e., "WHERE $R.f = value$"). The *end tables* of Q are those that appear in the SELECT clause of Q (i.e., "SELECT $R.f$"). Other tables of Q are referred to as *intermediate tables*.
2) Let $\mathcal{G}(Q)$ denote the subgraph of the schema graph corresponding to Q. A *parallel edge of size* q in $\mathcal{G}(Q)$, denoted $E = [e_1|e_2|\ldots|e_q]$, is a set of q distinct edges $e_i = R.f_i - R'.f'_i$, $1 \leq i \leq q$, between the same two tables R and R', such that all edges e_i appear in $\mathcal{G}(Q)$, and $\mathcal{G}(Q)$ contains no other edges between R and R'. If $q = 1$, E is a simple edge, denoted as $E = e_1$.
3) A *slice* in the subgraph $\mathcal{G}(Q)$, denoted $\pi = E_1 - E_2 - \ldots$, is a succession of parallel edges that contains no cycles, except possibly self loops, such that the first table on π is a start table of Q, and the last table on π is an end table of Q. We denote by $S(\pi)$ and $T(\pi)$ the start, resp. end, tables of π.
4) For any slice π in the subgraph $\mathcal{G}(Q)$, we say that Q *supports* π. The *support set* of π, denoted $Supp(\pi)$, is the set of all queries Q that support π.

For example, in Figure 1(a), the slice $\pi_1 = e_1 - [e_2|e_3] - e_4$ has $S(\pi_1) = S_1$ and $T(\pi_1) = T_2$, and contains two simple edges, e_1 and e_4, and one parallel edge of size 2, i.e., edge $[e_2|e_3]$. Its support set is indicated in Figure 1(b). By abuse of notation, we say that a table R belongs to a slice π, and write $R \in \pi$, if there exists an edge in π having R as an endpoint.

Our approach to constructing new join queries is to first decompose all log queries into slices, then recombine appropriate slices into new queries, as detailed below. We assume that the user provides the sets of start and end tables she is interested in, denoted $\mathcal{S}$, resp. $\mathcal{E}$. Our conjecture is that, given $\mathcal{S}$ and $\mathcal{E}$, the oracle computes queries whose graph structure is as simple as possible. Thus, we use this as a guiding principle for selecting slices. For example, if we have selected two slices (S_i, T_j, π_k) and (S_ℓ, T_m, π_n), then there is no need to select any slice connecting S_i and T_m, or S_ℓ and T_j (such slices might introduce additional edges). Similarly, when there are multiple slices between tables $S \in \mathcal{S}$ and $T \in \mathcal{E}$ with different support sets, we select only one such slice. We call this the *parsimony principle*.

In our analysis of a real life query log, we have encountered cases in which two distinct slices π_i and π_j have the same start, resp. end, table, as well as identical support sets. This means that, according to the evidence of the query log, the whole subgraph $\pi_i \cup \pi_j$ must be included in any query that contains $S(\pi_i) = S(\pi_j)$ among its start tables, and $T(\pi_i) = T(\pi_j)$ among its end tables. In this case, we call π_i and π_j *twin slices*, and always include them together in the solution. We formalize this in the definition of a valid query, as well as in the definitions of quality measures, below.

Definition 2: Let $\mathcal{S}$, resp. $\mathcal{E}$, be user-specified sets of start, resp. end, tables. A set of slices $J = \{\pi_1, \ldots, \pi_k\}$ is a *valid query* for the input $(\mathcal{S}, \mathcal{E})$ if the following four conditions are satisfied:

1) $S(\pi_i) \in \mathcal{S}$, $1 \leq i \leq k$, and $\cup_{i=1}^{k} S(\pi_i) = \mathcal{S}$;
2) $T(\pi_i) \in \mathcal{E}$, $1 \leq i \leq k$, and $\cup_{i=1}^{k} \pi_i \supseteq \mathcal{E}$;
3) For any $\pi \in J$, if there exists π' s.t. $S(\pi) = S(\pi')$, $T(\pi) = T(\pi')$, and $Supp(\pi) = Supp(\pi')$, then $\pi' \in J$;
4) The graph $\cup_{i=1}^{k} \pi_i$ is connected.

If J is a valid query for $(\mathcal{S}, \mathcal{E})$, and no subset $J' \subset J$ is a valid query for $(\mathcal{S}, \mathcal{E})$, we say that J is *parsimonious* for $(\mathcal{S}, \mathcal{E})$.

Note that we use slightly different conditions on the start and end tables. By requiring that $\cup_{i=1}^{k} S(\pi_i) = \mathcal{S}$, we ensure that the computed query J imposes conditions on all tables specified by the user. For end tables, the similar requirement would be that $\cup_{i=1}^{k} T(\pi_i) = \mathcal{E}$, i.e., each end table is selected in at least one slice. While natural, this condition can be relaxed by requiring that each end table appears in at least one slice π_i. We do this because of the parsimony principle. Recall that each slice π is obtained from log queries, and represents an *observed dependence* between $S(\pi)$ and $T(\pi)$. However, it also implies a *potential dependence* between $S(\pi)$ and any table $T' \in \pi$. In other words, any query containing π can be augmented to return values from table $T' \in \pi$, without changing the values returned from its other end tables. The third condition in Definition 2 says that if J contains slices that have twins, then J contains their twin slices as well, while the fourth condition ensures that J does not degenerate into a cross product. Because a query must be a valid query first, the condition on twin slices is given priority over parsimony.

In general, there are many ways of selecting sets of slices that satisfy Definition 2, so there are many valid queries for a given input. To determine the best answer, we need to define a quality measure over valid queries. However, as we illustrate in the example from Figure 1, defining a good measure is not straightforward. In the following, we consider several possible measures, which we divide into two categories: global measures, and local measures.

Global Measures: This class of measures considers the support sets of the slices in a valid query, and computes the cardinality of various regions in the Venn diagram of these sets. We explain how the different measures we consider affect the solutions computed from the slices in Figure 1.

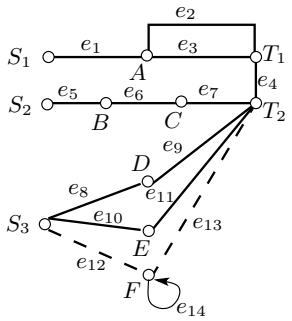
Example A: Assume that the sets of start and end tables are $\mathcal{S} = \{S_1, S_2, S_3\}$, resp. $\mathcal{E} = \{T_1, T_2\}$, and that the only slices between pairs of start and end tables are those depicted in Figure 1(b). The support set of each slice is indicated next to it. For example, slice $e_1 - [e_2|e_3]$ appears in queries Q_3 and Q_{12}. Note that query Q_3 supports both slices π_1 and π_2, i.e., it explicitly returns values from both T_1 and T_2. In contrast, queries Q_1, Q_2, Q_4 and Q_5 support only slice π_1.

There are many valid queries that can be constructed from $\pi_1, \ldots, \pi_{10}$. Examples include $J_1 = \{\pi_1, \pi_3, \pi_5\}$, $J_2 = \{\pi_1, \pi_3, \pi_7\}$, $J_3 = \{\pi_1, \pi_3, \pi_8\}$, $J_4 = \{\pi_1, \pi_4, \pi_5\}$, and $J_5 = \{\pi_1, \pi_3, \pi_4, \pi_7, \pi_8\}$. Which of these queries is the "best" answer? Query J_5 can be easily eliminated from consideration, as it violates parsimony. Indeed, since $J_2, J_3 \subset J_5$, then J_2 and J_3 are both more desirable than J_5. To compare the other solutions, we consider several measures, as follows.

One natural measure is the cardinality of the intersection of support sets: There are two queries, Q_3 and Q_4, that support all three slices in J_3. By contrast, the intersection of the support sets corresponding to J_2 is empty. Intuitively, J_3 is a better answer than J_2, since we have direct evidence in the log that it has been used before. Similarly, J_3 is better than both J_1 and J_4, whose intersection of support sets only contains Q_3. Hence, among the five example queries, J_3 is the best query according to the intersection measure.

Example B: Assume now that the path $S_3 - F - T_2$ does not exist in the schema graph. Hence, slices π_8, π_9 and π_{10} do not exist, and J_3 is not a candidate answer. According to the previous criterion, both J_1 and J_4 are better than J_2. However, by returning either J_1 or J_4, we are in fact ignoring the evidence of all queries $Q_1, Q_2, Q_4, \ldots, Q_9$. Note that 3 out of 7 queries in the support of slice π_7 also support slice π_1 (i.e., queries Q_1, Q_2, Q_5). Similarly, 4 out of 7 queries in $Supp(\pi_7)$ also support π_3 (i.e., queries $Q_6, \ldots, Q_9$). Thus, there is strong evidence in the log that slice π_7 is related to both slices π_1 and π_3, even though no query connects all three of them. By contrast, there is only one query, Q_3, that relates slice π_5 to π_1 and π_3 (or similarly, to π_1 and π_4). It is possible that Q_3 was issued for infrequent conditions on S_1, S_2 and S_3; or even that it was erroneously issued. In either case, it is reasonable to consider J_2 a better answer. The second quality measure we define maximizes the cardinality of the set of queries that support at least two slices in the answer.

A third possibility is to simply maximize the cardinality of the union of support sets. This ignores the importance of co-



(a)

Slice	Support Set
$\pi_1 = e_1 - [e_2\|e_3] - e_4$	$Supp(\pi_1) = \{Q_1, Q_2, Q_3, Q_4, Q_5\}$
$\pi_2 = e_1 - [e_2\|e_3]$	$Supp(\pi_2) = \{Q_3, Q_{12}\}$
$\pi_3 = e_5 - e_6 - e_7$	$Supp(\pi_3) = \{Q_3, Q_4, Q_6, Q_7, Q_8, Q_9\}$
$\pi_4 = e_5 - e_6 - e_7 - e_4$	$Supp(\pi_4) = \{Q_3\}$
$\pi_5 = e_8 - e_9$	$Supp(\pi_5) = \{Q_3, Q_{10}, Q_{11}\}$
$\pi_6 = e_8 - e_9 - e_4$	$Supp(\pi_6) = \{Q_3\}$
$\pi_7 = e_{10} - e_{11}$	$Supp(\pi_7) = \{Q_1, Q_2, Q_5, Q_6, Q_7, Q_8, Q_9\}$
$\pi_8 = e_{12} - e_{13}$	$Supp(\pi_8) = \{Q_3, Q_4\}$
$\pi_9 = e_{12} - e_{13} - e_4$	$Supp(\pi_9) = \{Q_3\}$
$\pi_{10} = e_{12} - e_{14} - e_{13}$	$Supp(\pi_{10}) = \{Q_4\}$

(b)

Fig. 1. Computing valid queries to connect start tables $\{S_1, S_2, S_3\}$ and end tables $\{T_1, T_2\}$: (a) schema subgraph; (b) slices and their support.

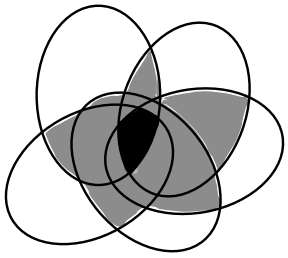


Fig. 2. Illustration of $\mu_I(J)$ (black region), $\mu_N(J)$ (black and grey regions), and $\mu_U(J)$ (black, grey, and white regions) for a query J defined by 5 slices.

occurrence of slices in log queries. By this criterion, query J_1 is the best. We summarize this discussion as follows.

Definition 3: Let $\mathcal{S}$, resp. $\mathcal{E}$, be user-specified sets of start, resp. end, tables, and $J = \{\pi_1, \ldots, \pi_k\}$ be a valid query for the input $(\mathcal{S}, \mathcal{E})$.

1) The *intersection measure* of J, denoted $\mu_I(J)$, is the cardinality of the intersection of support sets in J, i.e., $\mu_I(J) = |\cap_{i=1}^{k} Supp(\pi_i)|$.
2) The *non-outlier measure* of J, denoted $\mu_N(J)$, is the cardinality of the set of queries that appear in at least two support sets in J. Formally, $\mu_N(J) = |\cup_{1 \le i < j \le k} (Supp(\pi_i) \cap Supp(\pi_j))|$.
3) The *union measure* of J, denoted $\mu_U(J)$, is the cardinality of the union of support sets in J, i.e., $\mu_U(J) = |\cup_{i=1}^{k} Supp(\pi_i)|$.

Query J is *optimal* according to the intersection | non-outlier | union measure if it satisfies the following two conditions: (a) J is *parsimonious* for $(\mathcal{S}, \mathcal{E})$; and (b) $\mu_{\{I|N|U\}}(J)$ is maximal over all parsimonious queries for $(\mathcal{S}, \mathcal{E})$.

Figure 2 illustrates the three measures for the support sets of a fixed set J of $k = 5$ slices: μ_I is the number of elements that appear in all k sets, while μ_N is the number of elements that appear in at least two sets. From the discussion above, it is not clear which measure is better, i.e., which one better reflects how the oracle would answer. One could imagine two sets of slices J_1 and J_2 for which $\mu_I(J_1) < \mu_I(J_2)$, but $\mu_N(J_1) > \mu_N(J_2)$. Of course, there are other measures we could consider. For example, we could define a measure as the number of elements that appear in (at most, at least, exactly) $k/2$ sets of J; or a new measure μ as a vector of several measures. In this paper, we choose to explore only the "extreme" cases μ_I and μ_N. We include μ_U for completeness, although we do not expect it to lead to good answers (as indeed verified in our experimental results).

Local Measures: So far, our approach to defining quality measures has been holistic, i.e., a query Q counts towards $\mu_{\{I|N|U\}}(J)$ only if Q supports at least an entire slice of J (for $\mu_I(J)$, Q has to support all slices in J). We now propose an alternative approach, in which queries may have fractional contributions towards the measure if they contain at least an edge in J (as opposed to an entire slice).

Definition 4: Let $\mathcal{S}$, resp. $\mathcal{E}$, be user-specified sets of start, resp. end, tables. Let $\Pi = \{\pi_1, \ldots, \pi_n\}$ be all the slices generated from log queries, such that $S(\pi_i) \in \mathcal{S}$ and $T(\pi_i) \in \mathcal{E}$. For any edge $e \in \cup_{i=1}^{n} \pi_i$, let $\Pi(e) = \{\pi_i \in \Pi \mid e \in \pi_i\}$ be the set of slices in Π that contain e.

1) The *relevance of a simple edge* $e \in \cup_{i=1}^{n} \pi_i$ is defined as $rel(e) = \frac{|\cup_{\pi \in \Pi(e)} Supp(\pi)|}{|\cup_{\pi \in \Pi} Supp(\pi)|}$, i.e., it is the fraction of queries in $Supp(\pi_1) \cup \ldots \cup Supp(\pi_n)$ that contain edge e. The *relevance of a parallel edge of size* q, $E = [e_1|\ldots|e_q]$, is defined as $rel(E) = \frac{\sum_{i=1}^{q} rel(e_i)}{q}$. The *irrelevance* of E is defined as $\nu(E) = 1 - rel(E)$;
2) The *irrelevance* of a slice $\pi \in \Pi$, where $\pi = E_1 - \ldots - E_m$, is defined as $\nu(\pi) = (\sum_{i=1}^{m} \nu(E_i))/m$; i.e., $\nu(\pi)$ is the average irrelevance of a parallel edge in π;
3) The *irrelevance* of a valid query $J = \{\pi_1, \ldots, \pi_k\}$ is defined as $\nu(J) = \sum_{i=1}^{k} \nu(\pi_i)$.

A valid query J is *optimal* according to the irrelevance measure if $\nu(J)$ is minimum over all valid queries for $(\mathcal{S}, \mathcal{E})$.

For example, in Figure 1, $\nu(e_1) = \nu(e_2) = \nu(e_3) = \frac{1}{2}$, $\nu(e_4) = \frac{7}{12}$, and $\nu(\pi_1) = \frac{1}{3}(\frac{1}{2} + \frac{\frac{1}{2}+\frac{1}{2}}{2} + \frac{7}{12}) = \frac{19}{36}$. The intuition for defining the relevance of an edge as above is that highly instantiated join edges are more likely to be part of an ideal solution. Thus, by maximizing the average relevance of an edge in our solution, we want to "guess" as many of the edges in an ideal solution as possible. We still choose edges one slice at a time, as this insures that we connect the

start and end tables appropriately. The problem is formulated in terms of minimizing the total irrelevance of slices, rather than maximizing their relevance, so that it closely resembles a minimum weight set cover problem. We detail this in the next section. Finally, note that $\nu(J)$ is defined as a sum, rather than an average, of $\nu(\pi_i)$, because we want to penalize queries with many slices. If a query J is optimal under ν, it immediately follows that it is parsimonious.

III. Algorithms

As described in Section I, our overall approach is to first decompose existing log queries into slices, and then recombine the slices to form a new query. We detail the two phases below.

A. Slice Extraction

In a pre-processing phase, we identify for each log query Q its sets of start and end tables, according to the WHERE and SELECT clauses of Q. Note that the sets of start and end tables need not be disjoint. From each start table R_1 of Q, we execute a depth first search procedure in the graph associated with Q, and generate all combinatorially distinct simple paths that end in some end table R_2 of Q, $R_2 \neq R_1$. If Q contains self-joins, we also generate paths that contain self-loops: each self-loop is traversed at most once by a path, and there are no cycles of length larger than 1 on the path. For each path π thus generated, we keep track of the start table fields of R_1, the end table fields of R_2, and add Q to $Supp(\pi)$. For example, in Figure 1, query Q_4 contains a self-join on table F. Thus, when processing Q_4, we extract both slices π_8 and π_{10}, and add Q_4 to their support sets. By contrast, query Q_3 only supports π_8, but not π_{10}, because it does not have a self-join on table F. The support set of a slice is represented as a sorted vector of query id's for small sets, or as a bit vector for larger sets. Computing all support sets requires one linear scan of the query log.

B. Slice Recombination

The main challenge for this step is scalability. The larger the query log, the larger the number of distinct slices generated from it. Even for a user input of type $(\{R_1\}, \{R_2\})$, i.e., for a single start table and a single end table, the number of slices to consider may be exponential in the number of tables in the database. This is because a slice may traverse any of the $2^{|\mathcal{R}|}$ subsets of intermediate database tables between R_1 and R_2. Even if two slices traverse identical sequences of tables, the join edges between the same pair of tables may involve different columns. Then, the slices contain different parallel edges, and are therefore distinct. Nor can we bound the number of slices between two tables by the number of log queries. Indeed, a single query - which corresponds to a connected subgraph in the schema graph - may generate exponentially many slices between the same pair of tables (R_1, R_2), corresponding to the number of distinct paths between R_1 and R_2 in the subgraph (e.g., if the subgraph contains a large clique). Computing such a large number of slices is already a challenge for the extraction phase, but it can be executed offline and in batch mode (e.g., when the query log changes significantly). By contrast, the process of query generation via slice recombination is user-driven and must be executed fast.

To cope with this problem, we adopt the strategy of limiting the number of slices between the same pair of start and end tables. More precisely, we consider only the slices with highest support sets for each pair of start and end table fields, and choose the top k (usually, $k = 1$ or $k = 2$) in the dataset from which we compute a solution. Let Π denote the set of slices thus chosen. It is possible that Π does not contain any of the "ideal" (i.e., oracle-recommended) solutions. However, we conjecture that Π still contains solutions that have large overlap with an ideal one. Our aim is to discover one such solution. As we discuss in Section IV, this conjecture almost always holds for the real life query log over which we have run extensive experiments.

We use a greedy approach for computing valid queries that are local maxima under each of the measures defined in the previous section. The overall structure of the algorithm is presented in Figure 3. Given the user input $(\mathcal{S}, \mathcal{E})$, we start by extracting all slices π that have $S(\pi) \in \mathcal{S}$ and $T(\pi) \in \mathcal{E}$. The initialization procedure in Step 2 selects a constant number C of slices from Π that maximize the measure μ over all choices of C slices. The selected set J must satisfy the following restrictions: Each of the C slices in J covers at least one (start or end) table not covered by the other $C-1$ slices; the C slices form a connected subgraph; and no two slices in J are twins. We impose the latter condition so that μ is not computed over fewer than C distinct support sets. Once the initial set of slices is chosen, we expand it to include their twin slices in Step 3. In our experiments, we use $C = 1$ or $C = 2$.

We say that a table $R \in \mathcal{S}$ is *start-covered* by the current solution J if there exists a slice $\pi \in J$ so that $R = S(\pi)$; and a table $R \in \mathcal{E}$ is *end-covered* by J if there exists $\pi \in J$ so that $R \in \pi$. Note that, if a table R is in $\mathcal{S} \cap \mathcal{E}$, then R may be end-covered by J without being start-covered by it. The algorithm continues to extend J so that it eventually start-covers R. The reverse is not true: if the table $R \in \mathcal{S} \cap \mathcal{E}$ is start-covered by J, then it is also end-covered by it. The algorithm greedily selects a new slice at each step, so that at least one more table in $\mathcal{S} \cup \mathcal{E}$ is covered by the slice, and the measure μ is maximized. The slice selection is subject to the condition that the new partial solution J is a connected graph. We check this in Step 8.2, by verifying whether the set of tables that appear in J and the set of tables that appear in π have non-empty intersection.

Global Measures: The greedy framework in Figure 3 is used for each of the three global measures defined in Section II, by setting $\mu = \mu_I$, $\mu = \mu_N$, or $\mu = \mu_U$, respectively. The solution thus computed is not necessarily parsimonious. For example, suppose that $\mathcal{S} = \{S_1, S_2\}$ and $\mathcal{E} = \{T_1, T_2\}$, and assume that the algorithm proceeds as follows: It first chooses a slice connecting the pair (S_1, T_1), then it chooses a slice for (S_1, T_2), and finally for (S_2, T_2). If the slices for (S_1, T_1) and (S_2, T_2) are connected, then the slice for (S_1, T_2)

COMPUTEQUERY($(\mathcal{S},\mathcal{E})$, μ)
μ = quality measure;

1. Compute $\Pi = \{\pi \mid S(\pi) \in \mathcal{S},\ T(\pi) \in \mathcal{E}, \pi$ in top k slices between $(S(\pi), T(\pi))\}$;
2. J=INIT(Π, μ);
3. $J = J \cup \{\pi' \mid \pi'$ twin slice of some $\pi \in J\}$;
4. $\Pi = \Pi \setminus J$;
5. $SCov = \{S(\pi) \mid \pi \in J\}$;
6. $ECov = \{R \in \mathcal{E} \mid \exists \pi \in J\ s.t.\ R \in \pi\}$;
7. **while** $((SCov \neq \mathcal{S})$ or $(ECov \neq \mathcal{E}))$ and $(\Pi \neq \emptyset))$
8. Choose $\pi \in \Pi \setminus J$ s.t.:
8.1 $S(\pi) \notin SCov$ or $T(\pi) \notin ECov$;
8.2 $J \cup \{\pi\}$ is connected;
8.3 $\mu(J \cup \{\pi\}) = \max_{\pi'} \mu(J \cup \{\pi'\})$
(max is over $\pi' \in \Pi$ that satisfy 8.1 and 8.2)
9. $J = J \cup \{\pi\}$; $\Pi = \Pi \setminus \{\pi\}$;
10. $SCov = SCov \cup \{S(\pi)\}$; $ECov = ECov \cup (\pi \cap \mathcal{E})$;
11. **for** each $\pi' \in \Pi$ s.t. π' twin slice of π
12. $J = J \cup \{\pi'\}$; $\Pi = \Pi \setminus \{\pi'\}$;
13. $ECov = ECov \cup (\pi' \cap \mathcal{E})$;
14. **endfor**
15. **if** no π can be chosen in step 8
16. **return** J as 'incomplete solution';
17. **endif**
18. **end while**
19. **return** J as 'complete solution'.

Fig. 3. Greedy approach to computing valid queries that maximize μ.

can be eliminated, to obtain a smaller valid query.

For the intersection measure, parsimony of the final solution J could be enforced in a post-processing step, by eliminating unnecessary slices. Note that this does not decrease $\mu_I(J)$ (although it may increase it). However, if J is generated as a local maximum under μ_N or μ_U, then eliminating slices from J is undesirable, as it may significantly decrease $\mu_N(J)$ or $\mu_U(J)$. We therefore prefer to return the solution as computed by the greedy method, without enforcing parsimony.

Additional considerations: It is possible that in Step 8.3 there are multiple slices π that maximize the value $\mu(J \cup \{\pi\})$. If that is the case, there are two natural choices on how to continue: randomly choose only one such slice; or, choose all slices for which the maximum is attained. In our implementation we prefer the latter solution, for the following reason. If π and π' are twin slices, then $\mu(J \cup \{\pi\}) = \mu(J \cup \{\pi'\})$ for any global measure μ, by definition. Thus, if π achieves the maximum value in Step 8.3, then π' also achieves it. By adding all such slices to J at the same time, we eliminate the need for Steps 11–14. Of course, the reverse is not true: two slices that achieve the maximum in Step 8.3 are not necessarily twins (however, they are very likely to be twins). Since there is no natural way to decide, in only one greedy step, which one is the better choice, we adopt the strategy of including all such slices.

Local Measure: The optimization problem in this case can be formulated as a minimum weight set cover problem, where the set of elements to be covered corresponds to the start and end tables, and each slice corresponds to the subset of start and end tables it covers. However, because start tables and end tables are considered covered under different conditions (Definition 2(1) and (2)), we must formulate the set cover problem a bit more carefully.

We define an instance $(X, \mathcal{C}, wt)$ of the minimum weight set cover as follows (X is the set of elements to be covered, and $\mathcal{C}$ is the set of subsets of X from which the cover is chosen): For each table $R \in \mathcal{S}$, define an element R^s in X, and for each table $R \in \mathcal{E}$, define an element R^e in X. Note that if $R \in (\mathcal{S} \cap \mathcal{E})$, then there are two distinct elements, R^s and R^e, corresponding to R in X. To define $\mathcal{C}$, we assume for simplicity that there are no twin slices in Π (we later discuss how to modify the definition for twin slices). For each slice π, define a subset of elements $C_\pi = \{(S(\pi))^s\} \cup (\{R^e \mid R \in \mathcal{E} \cap \pi\})$, and let $\mathcal{C} = \{C_\pi \mid \pi \in \Pi\}$ be the set of all such subsets. In addition, we define the weight of the subset C_π to be $wt(C_\pi) = \nu(\pi)$. Then, $(X, \mathcal{C}, wt)$ is an instance of the minimum weight set cover problem, whose optimum corresponds to the optimal query under the local measure ν. Indeed, the goal of the minimum weight set cover is to choose a subset $\mathcal{C}^* \subseteq \mathcal{C}$ that covers X (i.e., $X \subseteq \cup_{C \in \mathcal{C}^*} C$) and for which $wt(\mathcal{C}^*) = \sum_{C \in \mathcal{C}^*} wt(C)$ is minimum over all such covers of X. Since $\mathcal{C}^*$ covers X, this implies that the set $J^* = \{\pi \mid C_\pi \in \mathcal{C}^*\}$ start-covers $\mathcal{S}$ and end-covers $\mathcal{E}$. To prove that J^* is minimal under ν, consider any other query J that is valid for $(\mathcal{S}, \mathcal{E})$. The set $\mathcal{C} = \{C_{\pi'} \mid \pi' \in J\}$ is a cover of X, which implies that $wt(\mathcal{C}^*) \leq wt(\mathcal{C})$, i.e., $\sum_{\pi \in J^*} \nu(\pi) \leq \sum_{\pi' \in J} \nu(\pi')$.

If Π contains twin slices, we modify the definitions of $\mathcal{C}$ and wt as follows. A subset $C_\pi \in \mathcal{C}$ corresponds to the tables covered by π, as well as any of its twin slices. More precisely, let $T(\pi) = \{\pi\} \cup \{\pi' \in \Pi \mid\ \pi$ and π' are twin slices$\}$. We define $C_\pi = \{(S(\pi))^s\} \cup (\bigcup_{\pi' \in T(\pi)} \{R^e \mid R \in \mathcal{E} \cap \pi'\})$. Note that, for any twin slices π and π', $C_\pi = C_{\pi'}$. Let $\mathcal{C}$ be the set of distinct subsets C_π. We also define $wt(C_\pi) = \nu(T(\pi)) = \sum_{\pi' \in T(\pi)} \nu(\pi')$.

The algorithm from Figure 3 is adapted to the greedy approach of set cover as follows. In Step 8, we choose a set of twin slices $T(\pi)$, rather than a single slice. We define the measure μ for a set of slices $J \cup T(\pi)$ as follows: Let $SCov(J), ECov(J)$ be the tables in $\mathcal{S}$, resp. $\mathcal{E}$. that are start-covered, resp. end-covered, by J. Similarly, let $ECov(T(\pi))$ be the tables in $\mathcal{E}$ that are end-covered by $T(\pi)$. Note that there is only one table that is start-covered by $T(\pi)$, i.e., $S(\pi)$. Then

$$\mu(J \cup T(\pi)) = \frac{|S(\pi) \setminus SCov(J)| + |ECov(T(\pi)) \setminus ECov(J)|}{\nu(T(\pi))} \quad (1)$$

With this modification, Steps 11–14 in Figure 3 become unnecessary. The following claim is immediate from the well known approximation result for minimum weight cover.

Proposition 1: Algorithm COMPUTEQUERY($(\mathcal{S}, \mathcal{E})$, μ), with μ as in Equation 1, computes a valid query for $(\mathcal{S}, \mathcal{E})$ of irrelevance at most $\nu_\Pi \log N$, where ν_Π is the minimum irrelevance of a valid query for $(\mathcal{S}, \mathcal{E})$, chosen from the set Π, and $N = |\mathcal{S}| + |\mathcal{E}|$ is the total number of start and end tables.

Let Π_{all} denote all slices with start tables from $\mathcal{S}$ and end tables from $\mathcal{E}$. Because $\Pi \subset \Pi_{all}$ contains only the top k slices for each pair of start and end table fields, the value ν_Π in Proposition 1 is different from the overall minimum

irrelevance of a valid query, denoted ν^*. This happens for two reasons. First, the optimal solution may contain slices that are in $\Pi_{all} \setminus \Pi$. However, because of the tight correlation between a small irrelevance and a large support set for a slice, we expect that most slices from the optimal query in Π_{all} are also contained in Π. Second, the irrelevance of an edge e can change when considering the support sets of all slices in Π_{all}, as opposed to the support sets of slices in Π. However, the irrelevance of a slice is an average over the irrelevance of its edges. Thus, for the change to be significant at the slice level, the majority of edges on the same slice must change their ν value significantly, and in the same direction (i.e., either increase or decrease). We expect that this is an infrequent occurrence. Overall, ν_Π is likely close to ν^*.

IV. Experimental Evaluation

We have conducted an extensive experimental study on a query log recorded over a period of 3 months in 2007 by a warehouse of several AT&T proprietary databases. All queries which accessed tables in a particular database were recorded, although queries typically involved joins across several databases. Many of the queries were generated by query generation tools working over an entity-relationship model, while others were issued as part of automated applications. Due to the length of time over which the log was recorded, we expect that it also contains human written queries. We did not have access to the schemas, nor to any metadata about the semantics of the databases. In particular, we did not have access to any of the query-generating tools or applications.

All our algorithms are implemented in Java, and run on an Intel Core 2 Duo PC with 2.33GHz CPU and 3.25GB RAM. For the remainder of this section, we denote by (**Global-I**, **Global-N** and **Global-U**) the greedy algorithms obtained from the framework in Figure 3, by setting $\mu = \mu_I$, $\mu = \mu_N$, and $\mu = \mu_U$, respectively. We denote by (**Local**) the algorithm obtained by setting μ as in Equation 1. We also implemented and evaluated two strawmen methods, denoted **Base** and **Random**, which we describe later in this section.

A. Evaluation Methodology

We consider each query Q in the query log itself to be an ideal solution for the input $(\mathcal{S}(Q), \mathcal{E}(Q))$, since we assume that the log is consistent with the oracle recommendations. Our algorithms are trained on part of the data. More precisely, there are 13,153 log queries that involve one or more join operations; these have between 1 and 6 start tables (average = 2.3) and between 1 and 9 end tables (average = 4). We randomly chose 20% out of them as test queries. We processed the remaining 80% queries, which are used as training data, for slice extraction, as described in Section III. This resulted in around 30,000 slices, which we stored along with their support sets on disk. We then recombine such slices as follows. For each test query Q, we compute its set of start tables $\mathcal{S}(Q)$ and end tables $\mathcal{E}(Q)$. We pass the input $(\mathcal{S}(Q), \mathcal{E}(Q))$ to our algorithm, and ask it to recombine slices extracted from the training data and compose a join query for that input. The result returned by our algorithm is compared to the original query Q for accuracy evaluation.

We repeated each experiment five times with independent choices of training/test data in each round, and present the average as the result in this section. The difference between the average and the result in any round is no more than 0.16%, demonstrating the statistical validity of our results.

For each pair of start and end tables, we find the top-k slices for each field pair in the start and end tables occurring in the logged queries, and merge these to obtain table level slices. We refer to these as the top-k candidate slices, and use only such sets when generating a solution using our methods. As we show in the experiments below, top-1 slices are usually sufficient to provide good results, and, depending on the measure μ, they may even yield better results than larger candidate sets.

Accuracy Measures: Let Q denote a test query and J denote the solution computed by one of the algorithms, for the input $(\mathcal{S}(Q), \mathcal{E}(Q))$. We denote by $Edge(Q)$, resp. $Edge(J)$, the set of all (simple) join edges that appear in the subgraph of Q, resp. J. To compare the quality of J with respect to Q, a natural approach is to evaluate the four measures defined below.

- The *precision* of J w.r.t. Q is the fraction of edges in J that also appear in Q, i.e., $p(J,Q) = \frac{|Edge(Q) \cap Edge(J)|}{|Edge(J)|}$;
- The *recall* of J w.r.t. Q is the fraction of edges in Q that are part of J, i.e., $r(J,Q) = \frac{|Edge(Q) \cap Edge(J)|}{|Edge(Q)|}$;
- The *jaccard coefficient* is $j(J,Q) = \frac{|Edge(Q) \cap Edge(J)|}{|Edge(Q) \cup Edge(J)|}$.
- The *f-measure* is $f(J,Q) = \frac{2 \times p(J,Q) \times r(J,Q)}{p(J,Q)+r(J,Q)}$

Thus, the precision indicates what percentage of edges in the computed result are actually part of the original query, while recall shows how many original edges are "discovered" by the algorithm. The jaccard coefficient is a natural measure for the similarity of two sets, while the widely used f-measure is the harmonic mean of precision and recall.

Confidence Measures: We also calculate, for each solution J, the measures $\mu_I(J)$, $\mu_N(J)$ and $\mu_U(J)$. Since these values can vary widely, depending on the particular input for which J was generated, we normalize each measure by $\mu_U(J)$, so we can plot the values for all solutions together. We call these normalized results confidence measures. More precisely, for each solution $J = \{\pi_1, \ldots, \pi_n\}$, we compute $conf1 = \frac{|\cap_{i=1}^n Supp(\pi_i)|}{|\cup_{i=1}^n Supp(\pi_i)|} = \frac{\mu_I(J)}{\mu_U(J)}$, and $conf2 = \frac{|\cup_{1 \le i < j \le n} Supp(\pi_i) \cap Supp(\pi_j)|}{|\cup_{i=1}^n Supp(\pi_i)|} = \frac{\mu_N(J)}{\mu_U(J)}$. The normalized μ_U is always 1, and we drop it. As the following experiments show, the greedy algorithm with $\mu = \mu_U$ is the least accurate, and is only included for completeness. In all the subsequent experiments, we report both $conf1$ and $conf2$ for all methods. However, note that Global-I does not attempt to maximize μ_N, and Global-N does not maximize μ_I.

Additional Methods: Our strategy of locally maximizing one of the measures defined in Section II aims to approximate the behavior of an oracle. A natural question that arises is whether less sophisticated strategies may do just as well.

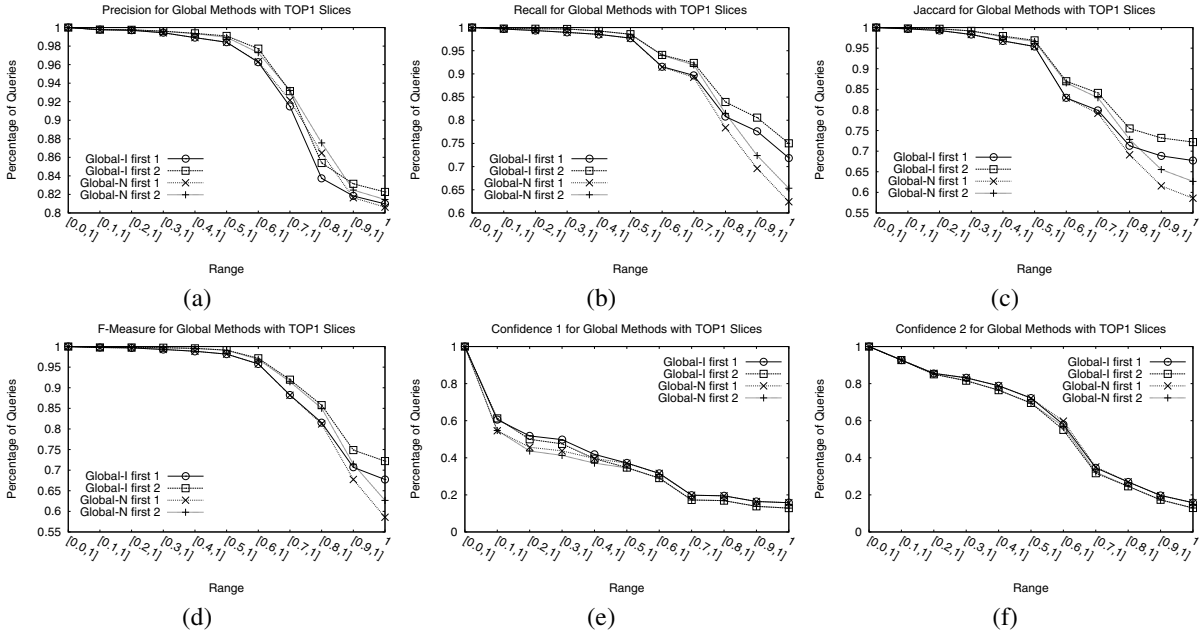


Fig. 4. Effect of Initialization Procedure on Global-I and Global-N with Top-1 Candidate Slice Set.

Given a set of start and end tables, are the slices we extract from the training data (i.e., the set Π in Figure 3) diverse enough to justify carrying on various selection procedures over them? Or is it the case that most slices in Π also appear in an ideal solution, and any simple strategy will give good results? To answer this, we implement two additional methods, denoted **Base** and **Random**. The Base method simply returns all potentially relevant candidate slices as the solution, i.e., $J = \Pi$. The Random method replaces the greedy step by a random choice; i.e., instead of computing the slice π that maximizes $\mu(J \cup \{\pi\})$ in Step 8.3 of Figure 3, it randomly picks a slice π from the current candidate set. However, each time a slice is picked, we verify whether it covers a start or end table that is not yet covered by the current solution J, and whether $J \cup \{\pi\}$ is a connected graph. If any of these conditions fails, π is discarded, and a new random choice is made. Hence, this is not a purely random method, as we still want to generate valid queries. In the following sections, we will evaluate and compare the performance of our proposed techniques with these two strawmen methods.

B. Efficiency

We measured the time to generate a join query recommendation in response to a user provided specification. Our greedy methods are very fast, and the average response time of these methods is 155ms using top-1 candidate slices, and 205ms using top-2 candidate slices. This response time includes computing the accuracy and confidence measures, and writing the results into files.

C. Evaluating Global Methods

In this section, we examine the performance of global methods, i.e., Global-I, Global-N, and Global-U. First, we look at the effect of the initialization procedure. As discussed in the previous section, the initialization procedure chooses the combination of C (non-twin) slices from Π that maximizes the measure μ. In our experiments, we set $C = 1$ or $C = 2$. Intuitively, we expect $C = 2$ to provide better results: Starting with the best set of two slices, instead of one, provides a higher probability of overlap with subsequent slices. In fact, the larger the value of C, the closer we are to examining all combinations of slices, and to generating the optimal solution. However, the running time of the initialization procedure is $O(|\Pi|^C)$, so large values of C are infeasible. We found that $C = 2$ provides a good tradeoff between accuracy and efficiency.

In Figures 4(a), (b), (c) and (d), we plot the distribution, over all test queries, of the values of precision, recall, jaccard and f-measure respectively. The solutions are computed by Global-I and Global-N from top-1 candidate slices (we obtained similar graphs using top-2 candidate slices). For Global-U, the choice of C is irrelevant: the set J initialized with $C = 2$ is the same as the set J obtained after initializing with $C = 1$ and executing one greedy step. The x-axis shows ranges of values for the accuracy measure, while the y-axis is the percentage of test queries for which the computed solution achieves a value in that range. The ranges on the x axis decrease as we move from left to right: the leftmost range is $[0, 1]$, while the rightmost range is $[1, 1]$. For example, in Figure 4(a), there are about 83% test queries for which Global-I returns a solution with precision equal to 1. As observed from the graphs, both Global-I and Global-N achieve higher precision, recall, jaccard and f-measure for $C = 2$, versus $C = 1$. An interesting phenomenon is that although using $C = 2$ yields more accurate results, it does not necessarily lead to higher values of $conf1$ or $conf2$. As shown in Figures 4(e) and

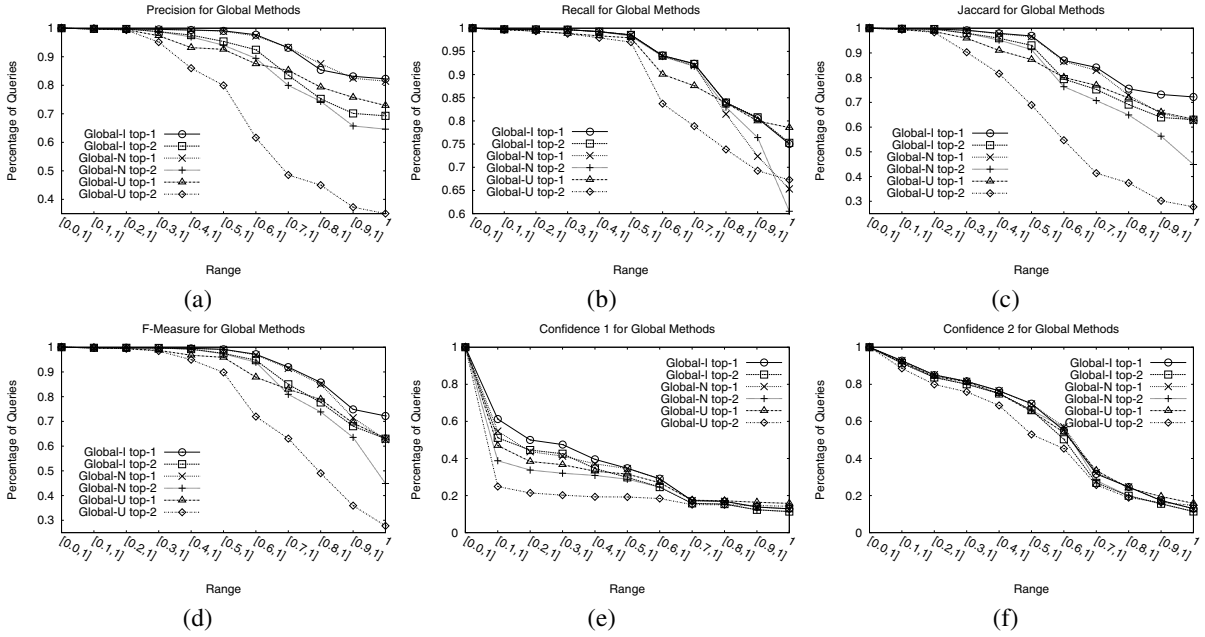


Fig. 5. Comparison of Global Methods on Top-1 and Top-2 Candidate Slice Sets.

(f), algorithms with $C = 1$ actually obtain slightly higher confidence. We will shed more light on the correlation between the accuracy and confidence measures later in the section.

Next, we compare the relative performance of the three global methods. For the graphs in Figure 5, we used $C = 2$ in the initialization procedure, which yields more accurate results for Global-I and Global-N. In this set of experiments, we include the results for each method running on both top-1 and top-2 candidate slice sets. The graphs in Figures 5(a), (b), (c) and (d) indicate that Global-I is the most accurate method, both for top-1 and top-2 candidate slice sets. It achieves the highest precision, recall, jaccard and f-measure among the three global methods. As expected, Global-U has the worst accuracy, with one exception: it achieves recall values equal to 1 on slightly more test queries than Global-N (Figure 5(b)). We conclude that maximizing the intersection of support sets for the slices in the solution is the best strategy to approximate ideal queries.

Figure 5 also shows how the choice of top-1 versus top-2 candidate slices affects the performance of the methods. Let Π_1, Π_2 denote the set of top-1, resp. top-2, candidate slices. All three methods perform better on Π_1 than on Π_2 for precision, jaccard and f-measure. For recall, Global-U performs better on Π_1, while Global-I and Global-N have similar performance on both candidate sets. To understand these results, we have examined several solutions in more detail. We found out that all three methods tend to select in the solution some slices from $\Pi_2 \setminus \Pi_1$. This is because of the greedy strategy: at some point, a slice $\pi \in (\Pi_2 \setminus \Pi_1)$ may be better for extending the partial solution. However, by the time the final solution is generated, that choice is not the best one we could have made. Although $\Pi_2 \supset \Pi_1$, and thus the optimal solution over Π_2 is at least as good as that over Π_1, the greedy strategy works better when it has fewer options. Note that Π_1 contains the slices with largest support sets for each pair of start and end tables, i.e., the most common and useful slices, and is thus sufficient for generating solutions for many test queries. We point out that for Global-U the quality of the results degrades significantly when replacing Π_1 with Π_2; while for Global-I and Global-N, although they also achieve better results on Π_1, the difference between Π_1 and Π_2 is relatively small. This suggests that Global-U is less resilient to having more options (which includes more bad options). The experiments confirm our intuition that maximizing the union of support sets, without considering their mutual intersections, is not a good strategy. By contrast, maximizing the common intersection, or the union of overlaps, guarantees that most slices do appear together in real queries.

D. Global Vs Local Methods

In this section, we compare the performance of the local strategy with that of global strategies. The results are shown in Figure 6. Since Global-I is the best among the global methods, we drop Global-N and Global-U from the graphs, for the sake of clarity. Although Local obtains high precision, similar to that of Global-I, its recall is much lower than that of Global-I on both top-1 and top-2 slices. This indicates that although Local selects relevant slices, it fails to find many of the slices in the ideal solution, which significantly impacts its recall. It is not surprising, therefore, that the jaccard and f-measure of Local are also worse than Global-I. However, Local performs better with respect to the confidence measures: it achieves $conf1 \in [0.5, 1]$ and $conf2 \in [0.9, 1]$ on more test queries than Global-I. This suggests that the slices returned by Local are strongly correlated, since a large intersection or

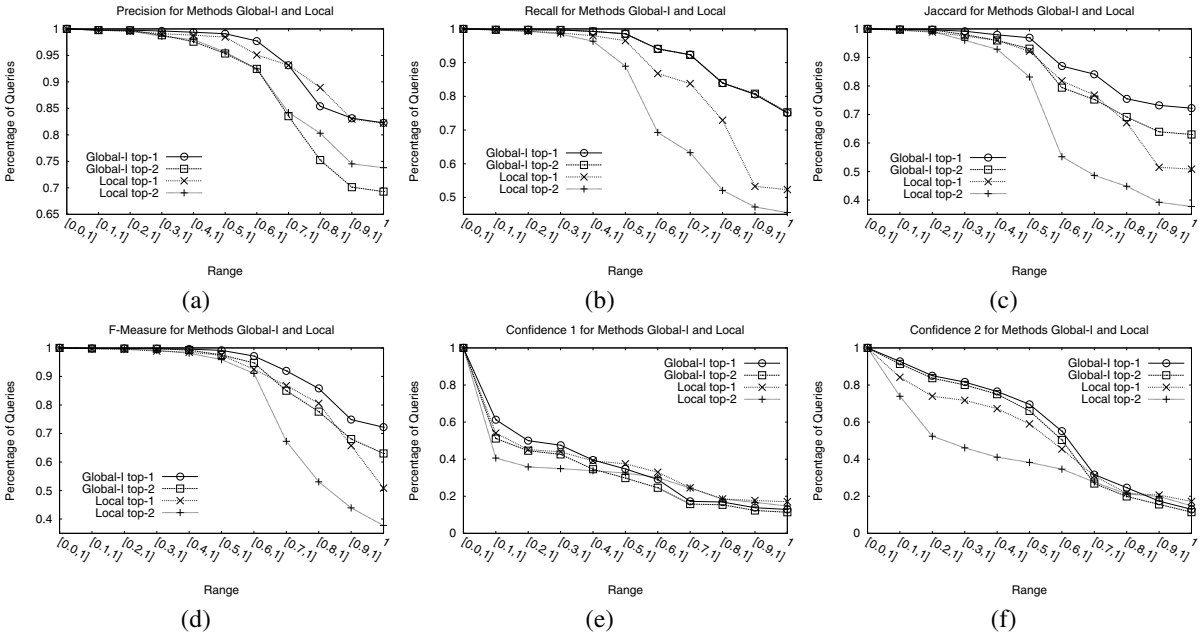


Fig. 6. Comparison of Global-I and Local on Top-1 and Top-2 Candidate Slice Sets.

overlap of support sets indicates that the corresponding slices appear together in many queries. However, such correlation among the final set of slices does not necessarily lead to high similarity between the solution and the original test query. Again, we defer discussion of this apparent disconnect between confidence and accuracy measures to later in the section. Sometimes, even when two test queries differ from each other only in one start or end table, or when the set of start or end tables of one test query is a subset of the corresponding set of the other query, they have significantly different sets of edges. Therefore, Local is less adaptive to small changes in the input than Global-I.

Although not plotted in Figure 6, we mention that Global-N outperforms Local on both top-1 and top-2 slices by achieving higher recall, jaccard and f-measure. Global-U beats Local only on top-1 slices, and degrades faster than Local on top-2 slices. The trends for the confidence measures are similar, with Local being better than the global measures.

E. Comparison with Base and Random

In this section, we evaluate the performance of Base and Random and compare them with our proposed algorithms.

First, we compare Random with Base by examining the influence of top-k slices on their performance. We discuss only the accuracy measures, as confidence measures are less relevant for these methods. From Figure 7, we see that the performance of both Random and Base degrades significantly on top-2 slices. The only exception is that, for the recall measure, Base performs slightly better on top-2 slices (Figure 7(b)). This is to be expected, since Base simply returns all candidate slices, and the more slices returned, the higher the probability that edges of the test query are covered by the result. However, the fact that the recall of Base is only marginally smaller for top-1 slices, versus top-2 slices (and in either case, it is almost always 100%), is very significant. It reinforces our conclusion that the greedy algorithms should be executed only over top-1 slices. Using top-2 slices does not significantly improve the potential recall of any solution, but it introduces many bad candidate slices. This is illustrated in Figure 7(a), where the precision for Base degrades on top-2.

Comparing the two methods, we note that, while Base achieves 100% recall almost all the time, the recall of results generated by Random is much lower. Although Random outperforms Base for precision on both top-1 and top-2 slices, the big gap of recall between the two methods makes Base better in the jaccard coefficient and f-measure on top-2 slices and for high values on top-1 slices. The conclusion is that Random fails to find as many accurate solutions as possible, although imposing the connectivity condition on the generated solution helps it achieve better results on top-1 slices.

We now compare Random and Base with our global and local methods. Since all methods perform better on top-1 slices, we restrict our experiments here to this setting. We use Global-I as the representative method for global measures, as it is the most accurate of the three. The results are depicted in Figure 8. Base and Random perform worse in terms of precision, while, as expected, Base achieves the highest recall. Local consistently performs worst among the four methods on recall, jaccard and f-measure, and we ignore it in the subsequent discussion. Although Base achieves 100% recall on almost all queries, while Global-I only achieves it in about 75% of the tests, the better recall of Base comes at a heavy price. By selecting all slices, Base generates larger, more complicated solutions, as reflected in its smaller precision.

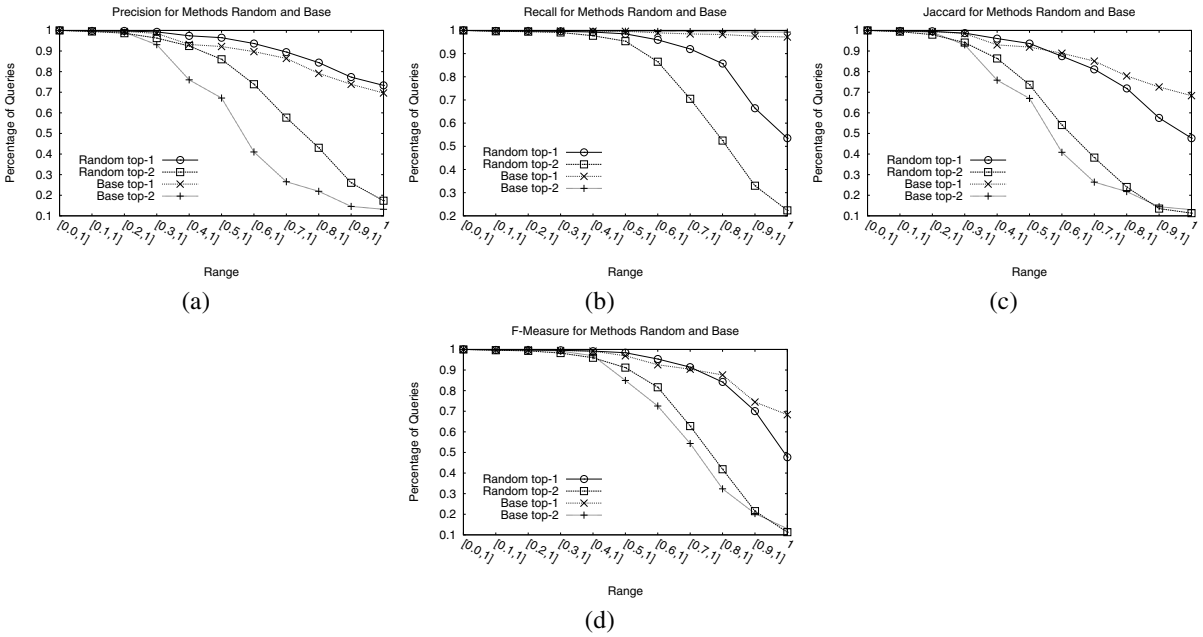


Fig. 7. Comparison of Random and Base on Top-1 and Top-2 Candidate Slice Sets.

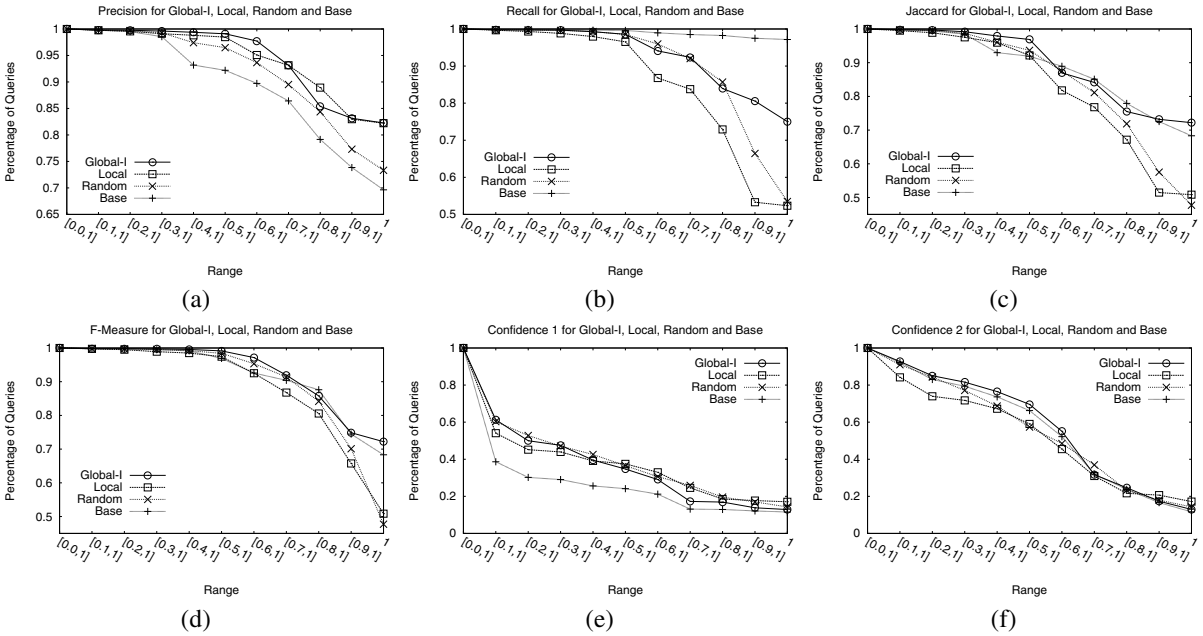
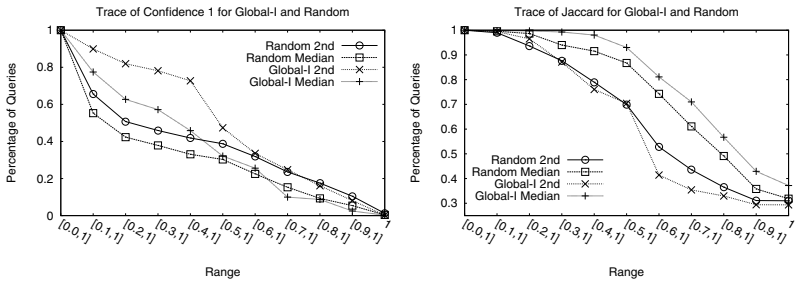


Fig. 8. Comparison of Global-I, Local, Random and Base on Top-1 Candidate Slice Sets.

Moreover, the precision of Base drops dramatically for top-2 slices (Figure 7(a)). We conclude that for many test queries, the sets of candidate slices contain bad choices, and that we need a strategy for pruning them away. We now compare the two competing strategies, i.e., Global-I and Random. Note that Global-I significantly outperforms Random for recall values in range $[0.9, 1]$ and equal to 1: Global-I achieves such recall values on approximately 14% and 22% more test queries than Random, i.e., it better approximates the ideal solution for around 360 to 570 tests queries. For jaccard and f-measure, Global-I outperforms Random in almost all cases. This shows that having a more sophisticated strategy for choosing slices is important, since naive random choices do not perform well. We further note that running Random without imposing the connectivity condition returned even worse recall results. Moreover, the recall, jaccard and f-measure of Random degrade significantly on top-2 slices (Figure 7), far more than the relatively mild accuracy loss of Global-I on top-



(a) (b)

Fig. 9. Comparison of Conf1 and Jaccard for Random and Global-I.

2 slices (Figure 5 (b)). We conclude that Global-I is both more accurate, and more consistent than Random.

Finally, we note that Random achieves slightly higher values for $conf1$ than Global-I does, even though Global-I locally maximizes the intersection of support sets, i.e., the un-normalized value corresponding to $conf1$. Yet again, this is at odds with the fact that Random achieves lower recall, jaccard and f-measure values. In the last part of this section, we turn our attention to better understanding the relationship between the $conf1$ and jaccard measures.[1]

F. Analysis of $Conf1$ and Jaccard

All the experimental results we have analyzed so far left us with an apparent paradox. Global-I consistently achieves a higher jaccard coefficient, but a lower $conf1$ value, than rival methods such as Local and Random. However, Global-I arrives at the solution precisely by trying to optimize the measure μ_I, which is the un-normalized version of $conf1$. For a complete picture, we have also run Global-I using $\mu(J) = conf1(J)$, instead of $\mu(J) = \mu_I(J)$ as the measure, but this only marginally improved the final reported value of $conf1$, while slightly decreasing the jaccard.

Since the final value of $conf1$ does not seem to correlate much with the jaccard coefficient, we hypothesized that intermediate values of $conf1$ must be better correlated. In other words, our conjecture is that for many tests, $conf1$ decreases significantly only in the final iterations of Global-I. This happens if there is no way to cover the last one or two tables without choosing a slice whose support set has small intersection with the current solution. However, while this reduces $conf1$, it may not be an inherently bad choice. The set of slices that describes the original test query itself may have very low $conf1$. What the jaccard measure indicates is that following the strategy of maximizing μ_I for each partial solution is correct, even though the final value may be small.

To test our hypothesis, we track the values of $conf1$ and jaccard for Random and Global-I each time a new slice is added to the solution. In Figures 9(a) and (b), we plot the value of $conf1$ and jaccard after the partial solution contains two slices (denoted 2nd), and after the partial solution contains half the slices of the final solution (denoted median) respectively. We observe that Global-I achieves much higher values for second and median $conf1$ than Random; while for jaccard, it's interesting to notice that Global-I achieves slightly lower values for second jaccard but much higher median jaccard than Random. Thus, we could conclude that $conf1$ in early iterations is strongly correlated with the eventual jaccard, and the greedy strategy of Global-I, especially in early iterations, is responsible for its overall good accuracy.

[1]Both jaccard and f-measure are more comprehensive measures than precision and recall. Since both show similar trends, we use jaccard as a representative here.

V. Conclusions

We have proposed a novel framework to automatically create join query recommendations matching minimal user specifications, based on an analysis of SQL query logs, inspired by ideas from program slicing. Our extensive evaluation of various quality measures, using the query log of an actual enterprise database system, shows that the best strategy for generating a solution (as a set of slices) is to locally maximize the measure μ_I of the partial solution. We show that we can generate good queries efficiently, by restricting the candidate set to top-1 slices. Our results demonstrate the practical viability of our approach for recommending join queries.

References

[1] R. Baeza-Yates and A. Tiberi. Extracting semantic relations from query logs. In *KDD*, 2007.

[2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[3] S. Chuang and L. Chien. Enriching Web taxonomies through subject categorization of query terms from search engine logs. *Decision Support Systems*, 35(1):113–127, 2003.

[4] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *SIGMOD*, 2002.

[5] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[6] Y. E. Ioannidis and Y. Lashkari. Incomplete path expressions and their disambiguation. In *SIGMOD*, 1994.

[7] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, 2006.

[8] G. A. Venkatesh. The semantic approach to program slicing. In *PLDI*, 1991.

[9] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[10] S. Wrobel. An algorithm for multi-relational discovery of subgroups. In *PKDD*, 1997.