Circumventing Data Quality Problems Using Multiple Join Paths

Yannis Kotidis
Athens University of Economics and Business
kotidis@aueb.gr

Amélie Marian
Rutgers University
amelie@cs.rutgers.edu

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

ABSTRACT

We propose the Multiple Join Path (MJP) framework for obtaining high quality information by linking fields across multiple databases, when the underlying databases have poor quality data, which are characterized by violations of integrity constraints like keys and functional dependencies within and across databases. MJP associates quality scores with candidate answers by first scoring individual data paths between a pair of field values taking into account data quality with respect to specified integrity constraints, and then agglomerating scores across multiple data paths that serve as corroborating evidences for a candidate answer. We address the problem of finding the top-few (highest quality) answers in the MJP framework using novel techniques, and demonstrate the utility of our techniques using real data and our Virtual Integration Prototype testbed.

1. INTRODUCTION

In any large organization, there are many database-centric applications, with overlapping features and functionality, ranging from sales and ordering tools to inventory and provisioning applications. These applications have authority over different pieces of data, and the difficulty of integrating legacy applications into a unified application for a given task typically results in the data being spread across multiple, autonomously managed databases. For instance, a multitude of ordering and provisioning tools can lead to customer accounts and billing data being present in different databases depending on, among other things, location, type of customer, etc. This fragmentation of data makes investigations across these databases problematic. A standard technique used for the task of querying across databases is the join path, linking two data fields, possibly in different databases, through intermediate data. Given a value for one of the data fields, a join path enables the identification of values reachable in the other field using the join path.

Compounding the difficulty of querying across databases is the prevalence of data quality problems, within and across databases (see, e.g., [6]). A typical phenomenon is the existence of duplicate, default and null values in columns of database tables that are supposed to be treated as primary/foreign keys, due to the inability to enforce integrity constraints across independent databases. For instance, a provisioning database may have a place-holder (i.e., a field) for storing customer contact information. However, often this field is empty (null values) or populated with dummy (default) values, since this information is of no immediate use for the application that deals with inventory and provisioning and which oversees this data. Data inconsistencies (e.g., multiple records with the same key value) are widespread, and can often be traced back to human errors, e.g., during manual data entry. Default values and data inconsistencies are examples of poor data quality prevalent in large databases.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CleanDB Seoul, Korea, 2006

1.1 VIP: Motivating Example

VIP is an integration platform, developed at AT&T, covering more than 30 legacy systems. It was developed in an effort to provide a platform for doing quick investigations and resolving disputes (due to data inconsistencies) between different applications.

A basic query that often arises in VIP is of the form "given the value of a field X, find the value of a field Y". For instance, when processing telecom data, an example query is: given the telephone number (TN) of a customer that shows up in a sales application (SALES), find the circuit id of the attached line. Since circuit ids are not part of SALES application, the users need to access the inventory application INVENTORY that can look up circuit ids using a provisioning order number (PON). Users have access to a front-end web interface that provides authentication and allows querying the underlying inventory dataset by pasting a single PON value into a form. The same front-end can also retrieve circuit information when queried using a TN, but the internal mapping is incomplete and contains inconsistencies. Thus, we need to devise additional strategies for locating the target circuit id by considering other applications that we may have access to.

By examining patterns of user interactions with the SALES, ORDERING, PROV and INVENTORY applications, we have been able to compute the schema graph, depicted in Figure 1, to help answer the query; the meaning of the numbers along the edges will be made clear when discussing our experimental results. PROV is an application that maintains provisioning records, while ORDERING is an ordering tool used primarily for small-business customers. Table 1 describes the fields depicted in the schema graph of Figure 1. The combined size of the databases behind these four applications is in the order of 100 million records.

The schema graph provides multiple paths to link a TN value in SALES to a CircuitID in INVENTORY. We list here a few of them:

- Using a TN value in SALES, we obtain PON values, based on the "intra-application" edge (SALES.TN, SALES.PON) depicted in Figure 1. We then access the INVENTORY application using these PON values and the "inter-application" edge (SALES.PON, INVENTORY.PON). There, we look up CircuitIDs using the

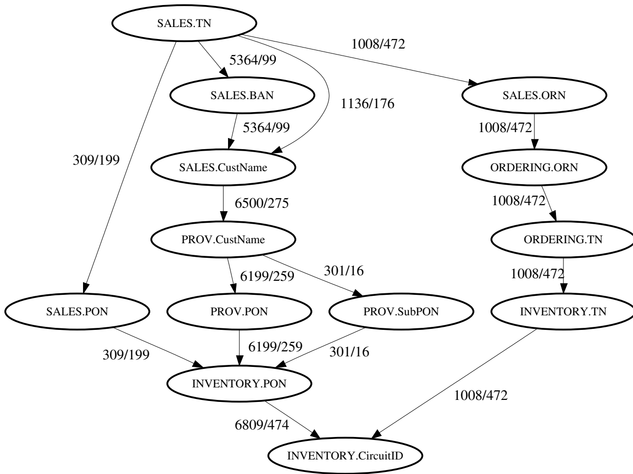

Figure 1: Schema graph for subset of VIP

Field Name	Description
TN	Telephone Number under investigation
BAN	Billing Account Number (primary key in biller)
CustName	Customer name (in biller and provisioning)
PON	Provisioning Order Number (key in provisioning applications)
SubPON	subsequent/related Provisioning Order Number (links multiple provisioning records for a customer)
ORN	Order Number (key in ordering applications)
CircuitID	Circuit the line is attached to

Table 1: Description of Fields in Figure 1

(INVENTORY.PON, INVENTORY.CircuitID) intra-application edge in INVENTORY. This corresponds to the left-most path in the schema graph.

- Given a TN in SALES, we can look up the customer name. This may be done directly, or via the billing account number for the customer. Notice that due to internal inconsistencies the two methods might give us different results. We can then input the customer name in the PROV application to retrieve all known PONs for the customer (from the PON and SubPON fields) which can be then used to probe INVENTORY, as in the first case. This corresponds to the set of middle paths in the schema graph of Figure 1.
- Small-business customers typically have multiple working telephone lines sharing the same circuits. For such customers, we can obtain the order number (ORN) in SALES, probe ORDERING and get all other lines ordered by the customer. Using this set of telephone numbers, we can probe INVENTORY multiple times. Even though, as explained, the internal TN-to-CircuitID mapping in INVENTORY is often incomplete, we can use the expanded set of all TNs in the customer order to try and find matching circuit ids in INVENTORY. This corresponds to the right-most path in the schema graph of Figure 1.

Given the different schema graph paths that link the TN input field (in SALES) to the CircuitID output field (in INVENTORY), which *join path* should be used to identify query answers?

1.2 Multiple Join Path Framework

When querying across multiple databases, in the presence of data quality problems, choosing any one join path results in missing answers, but choosing multiple join paths may lead to conflicting answers, especially when only a single answer is expected. Efficiency of query answering is also a concern. For instance, the return of a default value by an application may result in a significant number of probes to applications that follow it in a join path. Furthermore, different join paths that share edges need to be processed in a coordinated manner so that we avoid probing with the same input values multiple times.

The *Multiple Join Path* (MJP) framework proposed in this paper resolves these problems as follow:

- It takes *all* join paths in the schema graph into account.
- Each data path (schema path instance) is *scored*, taking the quality of integrity constraints (keys, functional dependencies), possibly across multiple databases/applications, and the quality of the data with respect to the integrity constraints into account.
- Multiple data paths between the same TN, CircuitID value pairs are treated as corroborating evidences, and data path scores are agglomerated to yield scores for CircuitID values.
- All join paths are considered when deciding the next application to probe. Intersecting data paths help re-use results of other join paths and reduce the number of probes to the applications.
- The top-few (typically 1) matches are returned as the desired answers. The schema graph and the computed data paths are used to prune unnecessary accesses to the applications.

When we are interested only in the top-few matches, it is extremely expensive to repeatedly probe the legacy applications, one schema graph edge at a time, to find all matching answers. This leads to the main technical problem addressed in this paper, the *Multiple Join Path Problem*:

> Given a schema graph identifying multiple join paths between field X and field Y, and a value $X = x$, find the top-few values of Y that are reachable from $X = x$ using the schema join paths.

The contributions of our paper are as follows:

- We introduce the MJP framework, and an agglomerative scoring methodology, to quantify answer quality in the presence of data quality problems arising due to integrity constraint violations in primary and foreign key columns, across multiple databases (Section 2).
- We develop novel techniques to limit the probing of legacy applications to efficiently compute the top-few answers to the MJP Problem. The agglomerative scoring methodology essentially renders previous mechanisms for computing top-k answers inapplicable for our problem (Section 3).
- Finally, we evaluate our techniques using real data and our VIP testbed. In particular, we demonstrate both the utility of the agglomerative scoring methodology in the presence of data quality problems, and the efficiency of our algorithmic techniques for computing top-few answers. In our real telecom example, we observe a reduction in the number of probes to the legacy applications by a factor of up to 18 in some cases (Section 4).

2. THE MJP PROBLEM

In this section, we introduce the Multiple Join Path framework, and our agglomerative scoring methodology, to quantify answer quality in the presence of data quality problems in multiple databases.

2.1 Queries and Answers

A basic query of interest is of the form "given the value of a field X, find values of a field Y", where X and Y refer to specific fields of individual applications. For instance, when processing telecom data, example queries include:

- Q1: given the telephone number of a customer (in SALES), find the circuit id (in INVENTORY) that the line is attached to.
- Q2: given a circuit id (in INVENTORY), find the customer names (in SALES) whose telephone numbers attach to this circuit id.

In the case of query Q1, one would expect there to be exactly one resulting answer. Since multiple telephone numbers may be attached to a circuit, query Q2 may have more than one answer. In both cases, X and Y are fields in different databases, so we need to establish *join paths* that link these two fields. There may be multiple possible join paths between any two given fields, and the schema graph, discussed next, identifies these possibilities.

2.2 Schema and Data Graphs

A schema graph is a 3-tuple (G, X, Y), where:

- $G = (V, E)$ is a *directed acyclic graph*, whose nodes $V = \{X, Y, \ldots\}$ are labeled by field names of accessible applications, and $E \subset V \times V$ are directed edges.
- $X \in V$ is the unique source (no incoming edges), and $Y \in V$ is the unique sink (no outgoing edges) of G.

A directed edge $(v1, v2) \in E$ is referred to as an *intra-application* edge, if $v1$ and $v2$ are fields in the same application; otherwise, it is an *inter-application edge*. A directed path P from X to Y in G is referred to as a *join path*. For instance, the schema graph of Figure 1 has six possible join paths from SALES.TN to INVENTORY.CircuitID, which can be used to answer query Q1. Thus, join paths in a schema graph identify different ways in which a basic query can be answered.

To ensure that join paths yield meaningful associations, not spurious correlations, we focus attention on the case where (i) all nodes in the schema graph (except, possibly, for source and sink nodes) are (possibly approximate) primary keys or foreign keys in their respective applications, (ii) inter-application edges correspond to (approximate primary key, approximate foreign key) associations, and (iii) intra-application edges are incident on an approximate primary key.

Given a specific value x of the source node X (e.g., telephone number, 555-5555, in query Q1), all join paths in the schema graph need to be explored to find all matching y values for the sink node Y (i.e., particular circuit ids). Intuitively, the data graph, defined below, captures these data associations. Given a schema graph (G, X, Y), a *data graph* is a triple (G_D, X_D, Y_D), where:

- $G_D = (V_D, E_D)$ is a *directed acyclic graph*, whose nodes V_D have labels of the form $T.A.v$, such that $T.A \in V$ and v is a value of field $T.A$, and $E_D \subset V_D \times V_D$ are directed edges such that $(T1.A1.v1, T2.A2.v2) \in E_D \Rightarrow (T1.A1, T2.A2) \in E$.
- $X_D \in V_D$ is the unique source of G_D, corresponding to value x of source node X of G, and $Y_D \subset V_D$ is a subset of the sink nodes of G_D, corresponding to values y_i of sink node Y of G.

For instance, given the schema graph of Figure 1, an example data graph is shown in Figure 2. There are two paths in this data graph from source node SALES.TN.555-5555 to the answer depicted by sink node INVENTORY.CircuitID.c1. Both these data paths correspond to the leftmost join path in the schema graph

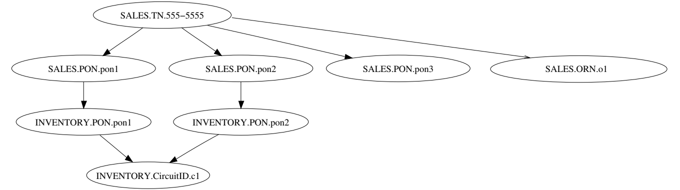

Figure 2: Data graph for query SALES.TN=555-5555

of Figure 1. Two additional nodes are present in this data graph, SALES.PON.pon3 and SALES.ORN.o1 (corresponding to schema graph nodes SALES.PON and SALES.ORN), which do not join with values in INVENTORY.PON and ORDERING.ORN, respectively. Note that the data graph can have multiple or no nodes corresponding to any specific node in the schema graph.

2.3 Scoring Answers

In a perfect world, the applications would have no internal data quality problems, and our basic query (given $X = x$, find Y) could be answered correctly by following all the join paths across the multiple applications starting from $X = x$, and taking the union of all the Y values that are reached along these individual join paths. But data quality problems are prevalent in large data-centric applications. For example, a primary key field (like the billing account number (BAN) field) may only be an *approximate* key [6]. Similarly, a functional dependency expected of an intra-application edge in the schema graph may be violated. As an example, we might find that the same telephone number is associated with two customer names due to manual data-entry errors in the SALES application.

So we are faced with the considerable challenge of answering our basic queries *without a priori knowledge of which values in the underlying databases are clean, and which ones are not*. To meet this challenge, we employ a probabilistic technique that scores data edges using values in the range $[0 \ldots 1]$. Thus, the score of a data edge $(T1.A1.v1, T2.A2.v2)$ represents our belief that the association between values $v1$ and $v2$ of fields $T1.A1$ and $T2.A2$ is correct. We will describe later how these scores are obtained. What is important is that this probabilistic interpretation of the scores allows us to combine scores across a data path.

Recall that a data path is just a sequential composition of data edges. Using a probabilistic interpretation of the data edge scores, assuming independence of the data edges in a data path, the *score of a data path* is defined to be the product of the scores of the constituent data edges. More formally, if $sc_1, sc_2, \ldots, sc_n$ are the scores of the constituent data edges of a data path P, then the score of P is given by:

$$sequential_com(sc_i, 1 \leq i \leq n) \quad = \quad \Pi_{i=1}^{n}(sc_i) \qquad (1)$$

As will be explained, this probabilistic interpretation assigns scores on data paths using data quality metrics on the edges. Thus, a high quality data path will get high scores independent of the length of the path, unlike, e.g., techniques like [1]. In fact, it is easy to see that the latter technique is just a special case of our framework when all data edges are scored with the same value in $(0, 1)$.

An answer may be corroborated by multiple data paths, and our scoring methodology agglomerates the scores of these data paths, using *parallel composition*, to compute the score of a Y value. For example in Figure 2 there are two data paths from SALES.TN.555-5555 to answer INVENTORY.CircuitID.c1. Different data paths are considered independent evidences and their scores are combined in a probabilistic manner. Formally, if $sc_1, sc_2, \ldots, sc_n$ are the scores of individual data paths $P_i, 1 \leq i \leq n$, between two nodes in the data graph, then, to ensure that all scores are in $[0, 1]$,

the score of the parallel composition of the P_i's is given by:

$$parallel_com(sc_i, 1 \leq i \leq n) \quad = \quad s1 + s2 - (s1 * s2) \quad (2)$$

where $s1 = sc_1$ and $s2 = parallel_com(sc_i, 2 \leq i \leq n)$.

Finally, the score of a Y value y_i is the score of the parallel composition of all the data paths from the source $X.x$ to the sink $Y.y_i$. This *agglomerative* scoring takes into consideration *all* the data paths that corroborate an answer.

Other combining functions may also be used without affecting the generality of the proposed methodology. The process that we describe in Section 3 requires the following two monotonicity properties, which allow for a broad selection of scoring functions:

- *Property 1:* the score of a data path is a non-increasing function of the scores of the constituent data edges.
- *Property 2:* the score of an answer is a non-decreasing function of the scores of the constituent data paths.

2.4 Data Edge Scores

Without a priori knowledge of the internals of the applications, or expertise on the quality of specific data items, our approach is to rely on expected functional dependencies between the exported data fields. For instance, in the telecom example, we expect a telephone number to uniquely identify a customer. Thus, when probing the SALES application, if we get two customer names for an assigned TN, this is a violation of an expected functional dependency and we should assign a lower score to the instantiated data edges.

Recall that intra-application schema edges $(T.A, T.B)$ capture associations where at least one of $T.A$ and $T.B$ is an approximate key in the corresponding application T. Assume, without loss of generality, that $T.A$ is the approximate key. Then the edge captures a *forward functional dependency* (FFD) from $T.A$ to $T.B$. Assume also that while answering a posed query, due to internal data quality problems, the following data edges are instantiated: $(T.A.v1, T.B.v11)$, $(T.A.v1, T.B.v12)$ and $(T.A.v2, T.B.v21)$. It is then obvious that the two different values $T.B.v11$, $T.B.v12$ associated with $T.A.v1$ are witnesses that $T.A.v1$ is in violation of the FFD and therefore data edge $(T.A.v2, T.B.v21)$ should have a higher score than edges $(T.A.v1, T.B.v11)$ and $(T.A.v1, T.B.v12)$.

Let $\{(T.A.v1, T.B.v1i), i = 1, \dots\}$ be the set of data edges instantiated for value $T.A.v1$ following this schema edge, and let $|.|$ denote the size of a set. To achieve the desired behavior, the score of each data edge $(T.A.v1, T.B.v1i)$ is set to:

$$sc(T.A.v1, T.B.v1i) = \frac{1}{|\{(T.A.v1, T.B.v1i), i = 1, \dots\}|} \quad (3)$$

The case when the schema edge captures a *backward functional dependency* (BFD) is handled symmetrically:

$$sc(T.A.v1i, T.B.v1) = \frac{1}{|\{(T.A.v1i, T.B.v1), i = 1, \dots\}|} \quad (4)$$

Finally, when both $T.A$ and $T.B$ are approximate keys, the edge captures a *symmetric functional dependency* (SFD) and the score is computed as:

$$sc(T.A.vi, T.B.vj) = \frac{1}{|\{(T.A.vi, T.B.*)\} \cup \{(T.A.*, T.B.vj)\}|} \quad (5)$$

where '*' means any value and is used to capture all data edges emanating from $T.A.vi$ (resp. leading to $T.B.vj$).

For an inter-application schema edge $(T1.A, T2.A)$, the score of a data edge corresponding to this schema edge is always 1, since the association between the fields is assured by the schema graph. An interesting extension is to consider *approximate matching* between values of field A in applications $T1$ and $T2$. In that case the score of the inter-application data edge is adjusted by using some notion of error metric (e.g., normalized edit distance or tf.idf for strings) between the values.

2.5 Multiple Join Path Problem

Our goal is to locate high quality information across multiple databases, in the presence of data quality problems. Since the different Y values that are reached from a given X value may have very different scores, we are interested only in the top-few matches. When we are interested only in the top-few matches, it is extremely expensive to repeatedly probe the legacy applications, one schema graph edge at a time, to find all matching answers, only to eventually discard the low scoring answers.

This leads to the main technical problem addressed in this paper, referred to as the *Multiple Join Path* (MJP) Problem:

> Given a schema graph identifying multiple join paths between field X and field Y, and a value $X = x$, find the top-few values of Y (with the highest scores) reachable from x using the multiple join paths.

Conventional top-k evaluation requires exact scores to be returned along with the matching answers, resulting in a ranking of the k results. In our agglomerative scoring methodology, since any unexplored data path could eventually corroborate a known Y value, resulting in a score increase (however slight), one would not be able to perform any early pruning for the MJP Problem, if one insisted on returning exact scores.

A more promising approach is where one can return top-k answers, where each answer is associated with a score range, and the result is a *set* of answers, not a ranking. In Section 3, we shall discuss novel solutions to the MJP Problem, and subsequently experimentally validate the utility and efficiency of our approach using real data and the VIP testbed.

3. THE MJP PROBLEM: SOLUTION

3.1 Incremental Data Graph Computation

Given a specific value x of the source node X in the schema graph, the data graph is initially instantiated with a unique (source) node $X_D = X.x$. For each newly inserted data node T_D in the data graph (excluding those in set Y_D of sink nodes), we create the set *open_edges(T_D)* to be the set of all schema edges $e \in E$ that emanate from the corresponding node T in the schema graph. As an example, for the schema graph shown in Figure 1 and for TN = 555-5555 being the TN in query $Q1$, the data graph is instantiated with a single node SALES.TN.555-5555. The set *open_edges(SALES.TN.555-5555)* will then include the following schema edges: (SALES.TN, SALES.PON), (SALES.TN, SALES.BAN), (SALES.TN, SALES.CustName) and (SALES.TN, SALES.ORN).

An *open node* in the data graph is any node T_D, not in Y_D, for which the set *open_edges(T_D)* is not empty. Our algorithms will proceed by carefully choosing an open node T_D and selecting one of the edges e in set *open_edges(T_D)* to explore. Following an intra-application edge $(T.A, T.B)$ for open node $T.A.u$ results in probing application T and retrieving a set of values for field $T.B$. For each unique value v_i of attribute $T.B$ in the result of this probe, we add a new node $T.B.v_i$ to the data graph and generate set *open_edges($T.B.v_i$)*. We further instantiate the data edge $(T.A.u, T.B.v_i)$ and compute its score. In Figure 3, we depict the data graph after exploring schema edge (SALES.TN, SALES.PON) for open node SALES.TN.555-5555. The application in this case returned three distinct values for SALES.PON: pon1, pon2 and pon3.

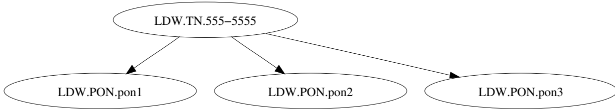

Figure 3: Data graph, after processing of edge (SALES.TN, SALES.PON)

Following an inter-application edge $(T1.A, T2.A)$ does not incur additional probes to the applications. Values of field $T1.A$ that do not appear in application $T2$ will not generate any new data nodes when a follow-up intra-application edge is processed. In either case, after edge e is explored it is removed from *open_edges(T_D)*.

We adopt a simple cost model that enumerates the number of probes to the applications while expanding the data graph to answer the user query. This cost model is reasonable in the absence of internal knowledge of the behavior of the applications.

3.2 Scheduling of Open Nodes

While building the data graph, we often have many open nodes to explore, each with at least one unexplored edge in *open_edges()*. We thus need a strategy that will lead to early pruning when computing top-k answers.

Since the data graph (G_D, X_D, Y_D) has a strong correspondence with the schema graph, we can pick the next open node/schema edge to explore using standard graph searching techniques like depth-first-search (DFS) or breadth-first-search (BFS) guided by the schema graph. Such techniques however are oblivious to the statistics we can collect both at the schema graph as well as at the (incomplete) data graph while processing the query. As is demonstrated by our experiments in Section 4, this results in substantially more probes to the applications. In what follows, we describe a greedy scheduling technique that is based on the notion of the *maximum benefit* of unexplored paths that go through open nodes.

Benefit computation involves two components. The first uses the statistics accumulated in the data graph to compute the score of all paths leading to an open node. The second component calculates the best way that the data graph can be augmented when following unexplored edges from an open node on the way to an answer. The fusion of these two components provides our benefit metric.

At each step, our algorithm maintains this benefit metric per open node/schema-edge in the data graph and schedules the next move using this metric. At an abstract level, our methodology for processing a user query can be summarized as follows:

- Start from the sole instance of source node X_D and expand one data node at a time. For any open node T_D, maintain the multiset of scores along data paths from X_D to T_D.
- By associating open node T_D with its schema node T, we can quantify the *residual benefit* of an unexplored schema edge e in *open_edges(T_D)* as the maximum possible contribution of the subgraph from X_D to any possible data node in set Y_D, passing through T_D using instances of e in the data graph.

As an example, we consider the data graph of Figure 3. For open node SALES.TN.555-5555 there are three unexplored edges in the set *open_edges(SALES.TN.555-5555)*: (SALES.TN, SALES.BAN), (SALES.TN, SALES.CustName), and (SALES.TN, SALES.ORN). Figure 4 shows the maximal subgraph that can be generated by exploring these edges in a way that maximizes the score of an answer. In this figure there are five paths from SALES.TN.555-5555 to schema node INVENTORY.CircuitID. For each path P_i, a schema edge is only instantiated once (since all edges are treated as FFD/SFD). However, a schema edge e' may generate one distinct

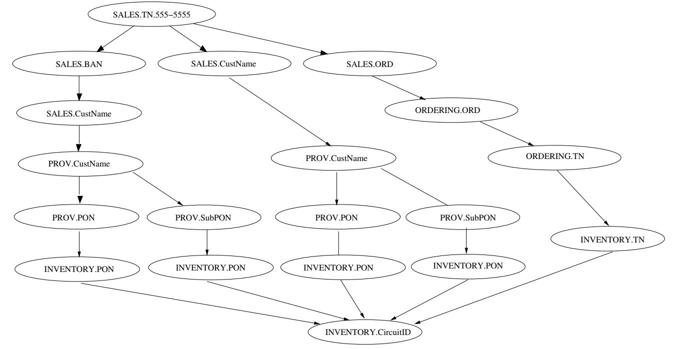

Figure 4: Maximum paths for unexplored edges of node SALES.TN.555-5555, after processing of edge (SALES.TN, SALES.PON)

data edge for each path P_i that contains e'. As an example, schema edge (PROV.CustName, PROV.PON) instantiates two distinct data edges in Figure 4.

Given one or more open nodes T_D in the data graph, we pick the next edge to explore as the one that maximizes our benefit metric. This is our *maximum benefit* policy, MAXB. In our experiments we see that MAXB outperform DFS and BFS, by a factor of up to 18:1.

3.3 Pruning Criteria

Unlike conventional top-k evaluation, where exact scores of answers are returned, for our MJP framework a more promising approach is to return the top-few answers, where each answer is associated with a score range. We distinguish between two versions of the problem:

- The exact top-k set $Y_D = (y_1, \ldots, y_k)$ is returned. For each answer y_i, we provide a score range $[s_{min}(y_i) \ldots s_{max}(y_i)]$.
- The top cluster of answers that is guaranteed to contain the top-k values is returned. Each answer is associated with a score range. We call this the *top-few* evaluation. Top-few is valuable when doing quick ad hoc investigations, since it allows for more pruning because of the weaker stopping condition.

Let $y \in Y_D$ be an answer present in the (incomplete) data path. Let $scores(y) = (sc_1, sc_2, \ldots, sc_n)$ be the scores of all data paths from X_D to y. Then, the minimum score of answer $Y = y$ is the parallel composition of the scores of all known paths to y:

$$s_{min}(y) = parallel_com(sc_i, 1 \leq i \leq n) \quad (6)$$

The maximum score of answer y is computed by additionally considering the maximum benefit of each open node and unexplored edge in the data graph, as discussed previously.

Through similar arguments we can compute the range of scores $[s_{min}(y_{unseen}) \ldots s_{max}(y_{unseen})]$ of an answer y_{unseen} that we have not encountered in our evaluation as $[0 \ldots max_contribution]$. The lower bound is trivial (when no new answer exists). The upper bound follows easily if we consider that all paths from the open nodes in the data graph terminate to a new answer y_{unseen}.

In a naive evaluation of the MJP Problem we stop when all open nodes in the data graph have been explored. However, one may stop earlier without exploring all open nodes, depending on the version of the problem. Assume set $\mathcal{Y}$ contains all answers that we have seen so far and also y_{unseen} (a placeholder for some answer we have not yet encountered). Thus, $\mathcal{Y}$=$Y_D \bigcup \{y_{unseen}\}$. We order the answers in $\mathcal{Y}$ using their minimum scores as $y_1, y_2, \ldots,$ where $s_{min}(y_i) \geq s_{min}(y_j)$ when $i < j$. This order also implies

$s_{max}(y_i) \geq s_{max}(y_j)$. If set $\mathcal{Y}$ contains more than k answers, we may stop further processing under the following condition:

- In top-k evaluation, we stop when $s_{max}(y_{k+1}) \leq s_{min}(y_k)$. That is, the upper bound on the score of the k+1'th candidate y value is no larger than the score of the current k'th candidate.
- In the top-few evaluation, we may stop if $s_{max}(y_{unseen}) \leq s_{min}(y_k)$. If this condition holds then any new answer cannot possible be scored higher that our current k'th candidate y_k. Thus, the top cluster is identified and we return those y_i's with $s_{max}(y_i) \geq s_{min}(y_k)$.

4. EXPERIMENTS

In this section, we experimentally evaluate our solution using our VIP testbed. Due to lack of space we provide detailed results for one query in our real dataset (query Q1, Section 2.1). Results for other queries between pairs of nodes in the schema graph of Figure 1 were similar. Our main experimental results can be summarized as follows:

- Real datasets have a multitude of data quality problems and no join path is immune to these problems. Using a fixed path or the maximum path for answering a query can lead to missing answers (low recall). That is why, in our MJP framework, all paths are considered.
- The data graphs can be fairly large (for instance when default values are encountered). Our scheduling techniques based on the maximum benefit metric achieve substantial pruning by eliminating a large number of candidate paths from evaluation.

The rest of this section is organized as follows. In Section 4.1 we illustrate that real applications are faced with significant data quality problems. When joining data across diverse applications, we typically find many answers, even when a single answer is expected (for instance a single CircuitID for a TN in Q1). Thus, ranking is required to help users identify the correct answer. In Section 4.2 we demonstrate that top-1 answers typically have several instantiated data paths leading to them and an agglomeration of their scores is needed. An important observation is that even join paths with small schema weights in their edges are useful in determining top-1 answers. In Section 4.3 we demonstrate that using our benefit metric results in substantially fewer probes to the applications, often by a factor of 1:18. Using the top-few execution model, this reduction is further increased by a factor of 2.

4.1 Nature and Quality of Data

We used traces of real user queries and obtained a random sample of 150 TNs that users ran investigations upon. We then used the schema graph to obtain circuit ids for these TNs (i.e., using k=∞). We noticed that there is a large number of TNs (56) that return no matching circuit ids. This is because (i) the INVENTORY dataset is incomplete and (ii) the provisioning key is often missing in SALES, forcing join paths either through customer names (CustName) or order numbers (ORN). The distribution is heavy-tailed, as there are many TNs for which we obtain 50 or more circuits through the schema graph. The maximum number of circuits returned for a single TN was 257. It is clear that most queries return a lot of answers. In fact only 2 TNs returned just one circuit! Thus, we need to be able to prune the long lists of matching circuit ids in order to provide meaningful answers to the user.

Using answer scores, we classify user queries into the following classes (in parentheses we show the number of TNs in each class).

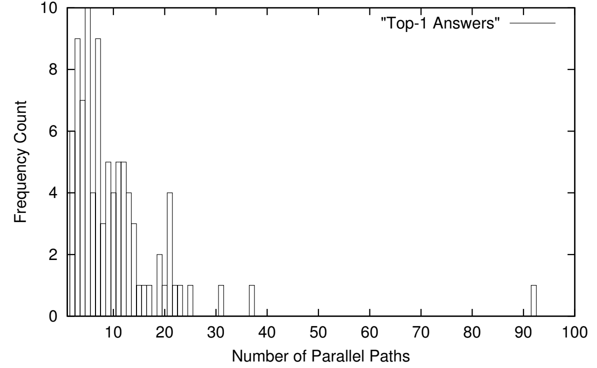

Figure 5: Number of parallel paths in top-1 answers

- **hH: heavyHitters(10):** These are the top-10 queries (TNs) ranked by the number of matching circuits in our data. TNs in that group returned between 128 and 257 circuits.
- **oL: oneLarge(47):** This is the subset of TNs that returned one circuit id with score at least 1% and zero or more circuits with scores less that this threshold.[1]
- **mL: manyLarge(4):** This set of TNs have at least 5 matching circuits with score at 1% or higher.
- **mS: manySmall(8):** This set of TNs returned at least 5 answers, while no answer had score greater or equal to 1%.
- **aA: anyAnswer(94):** All TNs with any matching circuits.
- **nA: noAnswer(56):** These are TNs for which no answer (circuits) can be obtained from the data.

4.2 Benefit of Agglomerative Scoring

We now address the utility of our agglomerative scoring methodology. In Figure 5 we plot the number of parallel data paths that contribute to the top-1 answer for each TN with a non-empty answer (set `anyAnswer`). For the 94 top-1 answers, there is an average of 10 parallel paths per answer (for a total of 946), out of which roughly 2.5 parallel paths per answer (for a total of 229) are significant (score of the path is greater than 10% of final score). In contrast, when looking at all answers for each TN there are on the average just 1.7 parallel paths contributing to each answer.

A natural question one may ask is whether all the schema join paths are really relevant, or if one of them dominates in its contribution to the final scores. In Figure 1, we annotate the schema graph edges with two numbers. The first is the number of data paths leading to an answer that instantiated this edge. The second number is for top-1 answers only. Some interesting observations on the nature of the data can be drawn by interpreting these numbers. First, paths that go through the SALES.PON node are more likely to end up in a top-1 answer: 199 out of 309 overall. Similarly, probing the ORDERING application leads to a top-1 answer in almost half the cases. In contrast, many paths that use instances of nodes SALES.BAN, SALES.CustName do not end up in top-1 answers. However, it is still beneficial to include these nodes in the schema graph. We notice that 275/946 top-1 paths (paths that result in a top-1 answer) go through instances of these nodes. If we remove these nodes from the schema graph along with all paths that

[1]The low value of the threshold has been chosen to capture as many potentially relevant answers as possible, given the scoring methodology.

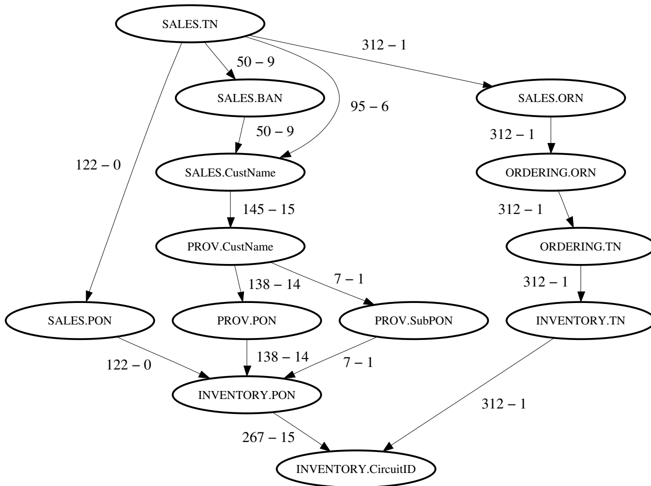

Figure 6: **Aggregate Path Statistics for top-1 answers (sets** `oneLarge` **-** `manySmall`**)**

		Top-1			Top-few		
	$k = \infty$	DFS	BFS	MAXB	DFS	BFS	MAXB
aA	246.9	245.4	125.1	47.1	244.9	97.3	24.6
hH	724.1	722.9	453.0	109.5	722.3	359.6	56.1
oL	261.0	259.1	104.6	14.6	251.4	95.5	14.1
mL	365.8	365.4	249.8	40.0	326.8	136.8	19.3
mS	258.5	258.5	253.8	184.8	258.5	231.7	119.6
nA	24.6						

Table 2: **Cost of top-1/top-few evaluation**

use them, then for the 94 queries with non empty top-1 answers, 8 return no result while for 77 the top-1 answer differs.

We next discuss the issue if there is any correlation between the join paths used and the class of queries. In Figure 6 we aggregate at the schema level the number of paths in top-1 answers for sets `oneLarge` (first number next to an edge) and `manySmall` (second number). The number of paths in set `oneLarge` is larger because more TNs belong to that set (47 compared to 8 in set `manySmall`). In the case of set `oneLarge`, paths through ORDERING schema nodes appear in more than half of the cases. However, again paths through the PROV dataset using customer names are significant in top-1 answers. In set `manySmall` there is only one top-1 answer with a path through ORDERING. Most of the paths (9 out of 15) go through the (SALES.TN,SALES.BAN) edge and subsequently to PROV, joining on the CustName attribute.

4.3 Efficiency of Top-k Evaluation

We use the number of probes to the applications to determine the efficiency of top-k evaluation. In Table 2, we present the average number of probes per TN during the top-k evaluation, for k=1. We also present the average number of probes when all circuits were requested (k=∞). For scheduling the next open node to explore, we tested the MAXB policy (discussed in Section 3) as well as the standard DFS and BFS orders obtained from the schema graph.

Top-1 evaluation, using MAXB, reduces the number of probes by a factor of more than 5, on the average, for TNs with matching circuit ids. A large number of queries (56/150) returned no answer, and for these queries top-1 evaluation cannot prune any paths. For the remaining TNs the greatest benefits arise for subset `oneLarge`. These are TNs for which one circuit stands out from

	Top-1			
	$k = \infty$	DFS	BFS	MAXB
aA	207/1189	206/1188	130/1109	59/679
hH	903/1189	902/1188	618/1109	162/679
oL	305/1189	304/1188	141/890	23/231
mL	415/793	414/793	327/625	52/83
mS	310/1128	310/1128	306/1106	237/629
nA	47/232			

Table 3: **Avg and Max Data Graph Size (edge set E_D)**

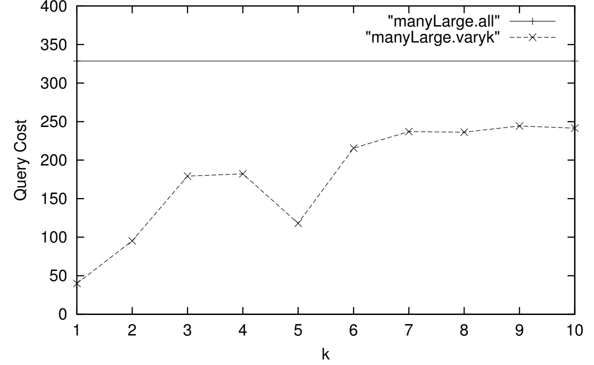

Figure 7: **Query cost, varying k (manyLarge set)**

the answer set and, thus, we can expect a lot of pruning during top-1 evaluation. Here, the MAXB policy reduces the cost of processing by a factor of 18:1 compared to the case where k=∞. For the set `manyLarge`, savings are smaller but still substantial (9:1). Even in the case of set `manySmall`, MAXB results in significant pruning. These are TNs for which a large number of circuits with very low scores were discovered. Looking at the instance graphs of these TNs we observed that most were due to a default value of field BAN in SALES, resulting in many matching customer names when following the intra-application (SALES.BAN, SALES.CustName) edge. Because of our scoring mechanism, all these paths were assigned very low scores and MAXB was able to prune a substantial number of them. In contrast, DFS for top-1 is almost as bad as getting all answers. This is because DFS follows deep paths through the schema graph to the end without concern of the current scores leading to an open node or potential benefits of open paths. BFS is slightly better since all paths are explored in unison.

In Table 3, we show the average (first number) and maximum (second number) size of the data graph for the same experiment. We notice that for k=∞ the data graph size is on the average 207 with a maximum instance of 1189 (edges). Thus, evaluation of Multiple Join Path queries in our framework has very modest requirements in terms of memory usage. We further notice that top-1 evaluation with the MAXB policy reduces these numbers by a factor of up to 13:1. This reduction of the data graph size will become significant in a multi-user environment when the server processes several queries at a time.

In Figure 7, we plot the query cost, varying k between 1 and 10. For comparison, we also show the cost when k=∞ (flat line). It is interesting that there is a drop in query cost for k=5. This suggests that the size of the top-cluster is, on the average, around five (circuits per TN) with the last 3 having similar scores. Thus, for k=3 or 4, additional queries are required to distinguish among them, while when k=5, we can stop earlier and report all of them.

In Table 2 we show the cost of the top-few execution, for k=1. The top-few execution, allows us to stop a query at an earlier stage,

when a superset containing the top-k cluster has been identified. As in the top-1 case, the MAXB policy by far outperforms the other alternatives. Comparing the numbers with the top-1 case, we see that we get a reduction in evaluation cost by a factor of two on the average. In most cases the number of answers returned to the user is very small, typically one. There are only 5 instances where we see more than 10 and all of them are for TNs with many small, indistinguishable, answers.

5. RELATED WORK

Scheuermann et al. [14] consider querying multiple database paths by allowing for some uncertainty in the attribute correspondences between databases in a multidatabase system. They return multiple query results ranked by some degree of confidence in the answer. However, to the best of our knowledge, our work is the first to take into account similar results from multiple paths as corroborating evidence and using this information to rank query results.

There has been much work in addressing the problem of identifying keyword query results in an RDBMS and ranking them based on some quality metric [8, 1, 2, 10, 9]. In such scenarios, the user queries multiple relations for a set of keywords and gets back tuples that contain all keywords, ranked by a measure of the proximity of the keywords. DBXplorer [1] and DISCOVER [10] use index structures coupled with the DBMS schema graph to identify answer tuples and rank answers based on the *number of joins* between the keywords. Our framework can also benefit from auxiliary structures like indexes and materialized views to speed up processing. BANKS [2] creates a data graph (a similar graph is used by [8]), containing all database tuples, allowing for a finer ranking mechanism that takes prestige (i.e., in-link structure) as well as proximity into account. Hristidis et al. [9] use an IR-style technique to assign relevance scores to keyword matches and take advantage of these relevance rankings to process answers in a top-k framework that allows for efficient computations through pruning. As with proximity search techniques, we consider all possible join paths and, as in [9], we want to allow for pruning of irrelevant data paths in order to speed up query execution time. A key difference with previous proximity search techniques is that none of these techniques deal with data quality issues, or agglomerate scores of multiple data paths that contribute to the same answer.

Top-k query evaluation algorithms that aim at identifying the k highest ranking answers to a query have been proposed for a variety of scenarios: multimedia [7, 11], web [12], expensive predicates [4], and RDBMS [3, 11]. Adaptive top-k strategies [4, 12] dynamically choose which operation to perform next based on current tuple scores and estimated statistics. In this paper, we use such adaptive techniques to select which join paths to investigate next. Most existing top-k techniques focus on cases where answer tuples can be mapped into a single relation, with all attributes values accessible through a unique ID, and rank the result tuples according to a predefined aggregation function (e.g., *min* or *weighted-sum*). While some of the proposed techniques [13, 11] apply to scenarios involving joins, and therefore deal with a potential explosion in the number of tuples, we are not aware of any top-k technique that does not consider each possible answer tuple as a single entity.

Traditional top-k techniques require exact top-k answer scores to be returned. In contrast, NRA [7], which only considers sorted accesses to multimedia sources, allows for the top-k answers to be returned as soon as they are identified, along with their possible range of scores. We use this relaxed stopping condition for our top-k evaluation, and present another efficient stopping condition: *top-few*, which returns a set of answers that are guaranteed to contain the best k answers.

Recently, Chaudhuri et al. [5] investigated the problem of ranking answers of database queries that are not very selective (*Many-Answers* problem) and propose a ranking function based on Probabilistic Information Retrieval ranking models. Our scoring functions also have a probabilistic interpretation and, similar to [5], ranking is proposed in order to prune a potentially large answer set. However, while in [5] the problem arises from loosely constrained queries, the complexity of our problems stems from (i) the existence of multiple join (schema) paths that can potentially link two attributes in the same or different databases, and (ii) low data quality that further increases the number of instantiated data paths for a given query. Approximating top-k answers, by offering guaranteed answer quality wrt the correct top-k scores [7, 4], or probabilistic guarantees [15], is an issue we do not address here.

6. CONCLUSIONS

This paper addressed the Multiple Join Path problem, of finding high quality query results that can be reached from a query node, by following one or more join paths in the schema graph, across multiple databases, in the presence of data quality problems. The framework proposed in this paper scores each data path that instantiates the schema join paths, taking data quality with respect to specified integrity constraints into account. Multiple data paths between the same nodes are treated as corroborating evidences, and data path scores are agglomerated to yield scores for matching answers. We develop novel techniques to efficiently compute the top-few answers within the Multiple Join Path framework, taking the agglomerative scoring mechanism into consideration. We evaluate our techniques using real data and our Virtual Integration Prototype testbed, and demonstrate both the utility of the agglomerative scoring methodology, and the efficiency of our algorithmic techniques for computing top-few answers.

7. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. *ICDE*, 2002.

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

[3] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM TODS*, 27(2), 2002.

[4] K. C.-C. Chang and S.-W. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. *SIGMOD*, 2002.

[5] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. *VLDB*, 2004.

[6] T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*. John Wiley, 2003.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *PODS*, 2001.

[8] R. Goldman, N. Shivakumar, S. Venkatasubramanian, H. Garcia-Molina. Proximity search in databases. *VLDB*, 1998.

[9] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. *VLDB*, 2003.

[10] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. *VLDB*, 2002.

[11] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB*, 2003.

[12] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM TODS*, 29(2), 2004.

[13] A. Natsev, Y. Chang, J. R. Smith, C. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. *VLDB*, 2001.

[14] P. Scheuermann, W.-S. Li, and C. Clifton. Multidatabase query processing with uncertainty in global keys and attribute values. *JASIS*, 49(3), 1998.

[15] M. Theobald, G. Weikum, R. Schenkel. Top-k query evaluation with probabilistic guarantees. *VLDB*, 2004.