AutoJoin: Providing Freedom from Specifying Joins
University of Iowa Technical Report 04-03

Terry Mason
University of Iowa
Computer Science
tmason@cs.uiowa.edu

Lixin Wang
University of Iowa
Mathematics
lixwang@math.uiowa.edu

Ramon Lawrence
University of Iowa
Computer Science
rlawrenc@cs.uiowa.edu

Abstract

SQL is not appropriate for casual users as it requires understanding relational schemas and how to construct joins. Many new query interfaces insulate users from the logical structure of the database, but they require the generation of valid joins. Our query inference system automatically infers potential joins. Although query inference has been studied for specific query languages or keyword searches, no general system for query inference on the relational model exists. In this paper, we present an algorithm that enumerates query interpretations at query time at least an order of magnitude faster than previous methods. The enumeration algorithm, called EMO, is combined with a method for detecting and eliminating semantically equivalent interpretations. The result is an approach that exhibits minimal overhead and reduces the number of queries that are considered ambiguous. Query inference is generalized to support complex queries that have user-specified joins, and the approach is usable with any relational schema and query interface. The algorithms are implemented in a query inference engine called AutoJoin. Experimental results demonstrate that query inference using AutoJoin can be efficiently performed on large, complex schemas.

1. Introduction

Despite significant improvement in the performance of database systems, the usability of database systems has not improved at a similar pace. Although many query interfaces, including conceptual and graphical languages, have been developed, they do not fully hide from users the complexity of SQL and the relational model. Thus, most end-user interaction with databases is through web and program interfaces that are costly to develop and maintain. Even worse, the size and complexity of database schemas are growing, especially as global schemas are constructed for integrated systems. Now, more than ever, users need a simpler method for querying databases.

One of the major challenges in generating SQL queries is using joins to connect concepts in different tables that were separated during the normalization process. Constructing these joins in SQL or through a graphical query interface is tedious, error-prone, and not intuitive to casual users [8]. Further, as schemas evolve, the joins in existing queries are often invalidated causing costly maintenance problems, especially for embedded queries. The impact of evolution can be minimized by specifying a high-level query (on attributes only) and then determining the required joins at query-time. Recent research using keyword searches on relational databases [7, 14, 1] allow the extraction of data without any required knowledge about the schema or metadata. These interfaces match keywords to data or metadata of a database, which then requires the determination of the joins to relate the keywords located in seperate relations.

The goal is to produce a query inference engine that dynamically converts an attribute-only or keyword query to SQL. For example, consider the query in Figure 1 on the TPC-H schema[1]. It is desirable if the query system could infer this query from the simpler attribute-only or keyword queries in Figure 3.

The challenge is to make query inference efficient, general, and practical for use in production databases. An ideal query inference system would be query language independent, so as not to restrict the types of queries that can be expressed to a particular query language. It is critical that the overhead of query inference be minimal even for very large schemas. Although ambiguity cannot be eliminated by a query inference strategy (since it is inherent in the relational schema), approaches to recognize and deal with ambiguity are necessary to make query inference valuable.

The original work on query inference can be categorized into two approaches: maximal objects [17, 22] and minimal cost [23, 26, 6]. Neither approach has been widely

1 http://www.tpc.org/tpch/

```
select P.p_name
from part P, nation N, partsupp PS,
    lineitem LI, orders O, customer C
where N.n_name = 'United States'
    and P.p_partkey = PS.ps_partkey
    and PS.ps_partkey = LI.l_partkey
    and PS.ps_suppkey = LI.l_suppkey
    and O.o_custkey = C.c_custkey
    and C.c_nationkey = N.n_nationkey
    and LI.l_orderkey = O.o_orderkey
```

Figure 1. TPC-H SQL Query

```
select P.p_name
where N.n_name = 'United States'
```

Figure 2. TPC-H Attribute-only and Keyword Queries

adopted for a variety of reasons, including that the algorithms are based on either the universal relation model [17], ER model [22, 23], User Viewpoint [6], or Semantic Graph [26], all of which require semi-automatic construction from the relational schema being queried. The high overhead of the approaches is another limitation. The maximal object approach pre-computes maximal sets of lossless joins in an inefficient grow all ways manner and provides no algorithms for handling large numbers of maximal objects. The cost-based approaches either perform a Steiner tree calculation on ER graphs as partial two trees which identifies only the lowest cost query or an exhaustive search on a limited portion of the graph for each query. Each approach limits the queries that can be inferred. Maximal objects only allow queries with lossless, natural joins. The cost-based approach returns either the lowest cost query interpretation or a query on a limited scope of the database. Finally, all of the approaches are restricted to a particular query language or model, which limits what queries can be inferred to the expressiveness of the query language or model.

Our overall contribution is demonstrating how to perform query inference efficiently, automatically, and independently of particular query languages on large database schemas. We have developed a query inference engine called AutoJoin that extends the previous approaches and improves the efficiency of query inference. The individual contributions are:

- An algorithm called EMO (for Enumerate Maximal Objects) that efficiently constructs all maximal objects (semantic interpretations) in a schema. EMO significantly outperforms previous approaches that fail on large schemas.
- A method for reducing the number of ambiguous queries by detecting and removing semantically equivalent interpretations.
- Efficient algorithms for processing query interpretations at query-time.
- An extension of the maximal object approach to handle queries with one lossy join.
- A graph representation of potential joins in a schema, called a *join graph*, that subsumes previous work and can be automatically built from the relational schema.
- A performance study that demonstrates the scalability of the approach.
- A discussion of the sources of ambiguity in schemas and how ambiguity can be handled.

Part 'United States'

Figure 3. TPC-H Keyword Search

The rest of this paper is organized as follows. Section 2 provides background on previous work on query inference. Section 3 presents the AutoJoin query inference engine that uses join graphs (Section 4) for representing joins in the relational schema and uses the EMO algorithm (Section 5) to rapidly enumerate all maximal objects (query interpretations) in the schema. Section 6 provides an algorithm for eliminating redundant maximal objects. An explanation on how query inference is implemented in AutoJoin is in Section 7. Extending maximal objects to handle lossy joins is covered in Section 8. Schema ambiguity and its effects are discussed in Section 9. A performance study in Section 10 shows that query inference can be performed with minimal overhead even for large schemas. The paper then closes with future work and conclusions.

2. Background

Query inference has been previously defined as "the translation of a query in a query language into an unambiguous representation of the query." [23] It is important to distinguish query inference from a query interface. We will use the terms *query interface* and *query language* interchangeably to mean a system for entering queries either in text or graphical form. A query interface or language may use an abstraction of the relational model to simplify query formulation. Various abstractions have been used such as the universal relation [18], QBE [28], the ER model [3, 20], and natural language [2]. The abstraction and the user interface improve the user's ability to formulate queries quickly and accurately. Commercial products such as the query builder in Microsoft Access use graphical interfaces to simplify querying. Surveys of query interfaces can be found in [8, 9].

Although abstractions reduce the complexity in join specification, they may still require the user to specify **all** joins required for the SQL query. For instance, both QBD* [3] and CQL [20] allow users to formulate queries by graphically navigating an ER model. A join is inserted in the query as a user traverses an edge from one entity to another. QBD* forces the user to connect all concepts via navigation, while CQL does not. Thus, only CQL requires query inference. Each of the keyword or token interfaces requires the inference to determine the joins for the intended query. Interfaces that can automatically infer joins and explain query interpretations are more usable.

None of the existing inference approaches removes equivalent join paths from the model of the schema in order to infer only ambiguous queries and avoid generating different queries with equivalent semantic meaning and results. Without the identification and removal of equivalent join paths, the inference methods will find seemingly different query interpretations that actually have the same semantic meaning, yielding the exact duplicate results. Maintaining only one join path of a set of semantically equivalent join paths simplifies the inference problem.

The universal relation model [18] requires query inference. Since the database is stored as a set of normalized relations instead of a single relation, the system has to determine how to join the relations to answer the user query. For schemas with acyclic graphs [5], there is only one way to connect the relations. However, for schemas with cyclic graphs, some queries are ambiguous. Maximal objects [17] were proposed to handle this ambiguity. A *maximal object* is a maximal subtree of the database graph that contains only lossless joins. The database graph is a hypergraph where the nodes are attributes, and each edge represents a relation. Before query time, maximal objects are built by growing larger objects from the initial objects (relations). An object can "grow" by merging it with another set of attributes (object) if the two sets of attributes can be connected using a lossless join. A join of sets of attributes R and S is lossless if $R \cap S \rightarrow\rightarrow R - S \mid S - R$. The algorithm [17] grows maximal objects from initial objects in all possible ways.

At query-time a user specifies a query on attributes that are mapped to maximal objects. If no maximal object contains all attributes, an error is generated as the query cannot be answered using only lossless joins. If a single maximal object contains all attributes, the joins required for the query are present in the maximal object. The maximal object may be pruned to remove joins connecting attributes in the maximal object that are not requested in the query. If more than one maximal object contains the attributes, then the query may be ambiguous with more than one lossless interpretation. The default action is to execute all unique interpretations (queries) and union the results.

One problem with the maximal object approach is that the growing process is very inefficient and requires explicit knowledge of functional and multivalued dependencies. Another limitation is that only queries with lossless joins are supported. Finally, the universal relation model is not used by designers and is a difficult abstraction for large schemas as it is hard to represent attribute roles and find unique attribute names.

The maximal object approach can be applied to ER graphs [22, 27]. The ER model is more familiar to database designers and can represent roles and additional semantics explicitly. The algorithms are modified to operate on undirected ER graphs instead of hypergraphs. Lossless joins are detected on ER graphs using relationship cardinalities (1:1,N:1) instead of functional dependencies. Although the ER model abstraction is an improvement over the universal relation, it is also not an ideal abstraction for casual users [8]. Both the UR and ER models are not directly extracted from the relational schema, so an administrator must either annotate or construct the appropriate model for either approach to be used.

The cost-based approach to query inference originated with the determination of the *minimal cost* join tree defined as the correct interpretation of the query. Wald and Sorenson [23] represent the ER diagram as a directed, weighted-graph with edge costs based on the cardinality of the relationships. An edge that traverses a lossless join (1:1,N:1) is assigned a cost of 1, and an edge that traverses a lossy join (1:N) is assigned a high constant cost. A user query is mapped to a set of nodes in the ER diagram. The inference problem is reduced to the minimum directed cost Steiner graph problem, which yields a single specific logical query based on the lowest cost join graph. The Steiner problem is NP-hard in general, although there is a linear time algorithm for database schema graphs that are partial two trees [23]. This approach will first attempt to generate the lossless interpretation with the fewest edges then default to the minimal interpretation with lossy joins if no lossless interpretation exists.

The Visionary interface [6] provides users a view of the data using left outer joins from a primary relation. Secondary relations are added to the view with left-outer joins to maintain all data. Users are involved in choosing which ambiguous paths to follow in order to remove the ambiguity in the User Viewpoint. The Viewpoint grows through foreign keys to primary key relationships. Large schemas or large data sets can lead to large Viewpoints which limits the effectiveness of the system.

The high level concepts represented on a semantic graph model [26] ranks the minimum cost interpretations through an exhaustive search algorithm with pruning. As the general search problem is NP Complete, the scope of the search is

constrained by limiting the number of joins and limiting the number of interpretations generated. While the approach allows for more complex queries, it ignores viable query interpretations by limiting the number of interpretations. The lossless property maintained in all previous approaches is ignored in calculating the values of these query interpretations, as the graph is not directed and lossy joins are treated the same as lossless interpretations.

Conceptual Query Language (CQL) [20] is a cost-based approach that finds the minimum path between specified concepts. The path enumeration algorithm for these abbreviated queries is from [25]. Since there may be many paths, CQL only selects the minimum path again ignoring lossy and lossless charcteristics. A similar system [10] enumerates paths. Natural language interfaces [2, 21] often require two forms of query inference. If the user can specify a query with an unconstrained vocabulary, then the system must infer what attributes and relations are being referenced in the query. The second form of inference, which is the one studied in this paper, is that given the attributes and relations referenced in the query, the system must infer the joins required to connect all concepts.

Motro [19] documents the challenges of using tokens or keywords to query a relational database. Data values and metadata values can be listed to query a database with no assumptions on the structure (schema) of the database. A dependency graph represents attributes and domains as nodes with links for functionally dependent attributes and attributes to its domain. The determination of all potential relationships between specified tokens is known to be NP Complete. Motro assumes a valid method to determine the potential joins. This is exactly where our inference engine fits into query interfaces for relational databases. The ultimate solution would be to identify the exact query intended by the user, but this is not possible if the solution has more than one interpretation. Ultimately, he proposes interation with the user to disambiguate the query. Intuitively, the interpretation with the fewest joins has the best chance to represent the intended join.

Each of the keyword search interfaces ranks the results based on specific cost metrics. The BANKS [7] keyword interface requires that the entire database be maintained in memory with links between all related tuples. This strategy requires inference at the tuple level with weights assigned to each tuple. The graph has tuples as nodes and foreign key to primary key links. This large, complicated graph limits scalability, as the tuples of the entire database must fit it into memory.

Discover [14, 13, 4] relates keywords located in seperate relations by growing all ways from a relation containing one of the keywords to all other relations with a limited number of possible joins between relations with keywords. This creates all possible graphs (potential interpretation) that meet this requirement. A pruning step is required to remove graphs that do not contain all of the keywords. This lead to cases where an order of magnitude extra graphs are created beyond the number of valid interpretations for a query. Precomputing the potential joins prevents searching paths with no potential of becoming a valid interpretation.

Keyword search DBXplorer [1] requires inference at the schema level. It identifies the location of keywords through an efficient symbol table and then infers the joins required for a query interpretation including all keywords by generating spanning trees.The direction of joins is ignored allowing lossy joins. All of the computation is executed at query time. They return all results ranked by minimum number of joins based on the proximity of keywords [12].

None of the approaches for keyword searches takes into account the lossless property of schemas. It can be argued the lossless joins better represent the intended relationships of the entities in the database design and are worth generating first. As an example consider a keyword query **Intel IBM** on the TPC schema, where `Intel` maps to multiple supplier locations in the *Supplier* table and `IBM` maps to multiple locations in the *Customer* table. The purchase computer hardware. The least amount of joins for an interpretation would produce a cross product of these tuples, which share a nation. Suppose Intel has 5 production facilities (in *Supplier* relation) located in the United States and IBM has 10 locations (in *Customer* relation) located in the United States. Fifty of the top results would be the cross product of these *Customer* locations in the US with *Supplier* locations in the US. The lossless interpretation would return the `IBM` locations with orders from related `Intel` locations. While suppliers and customers that share a nation is a valid interpretation, it could be argued that the stronger relationship between a customer and a supplier is through an order represented by the lossless interpretation even if it involves an additional join.

There are two limitations of cost-based approaches. First, they return the minimum cost interpretations which may not always be the interpretation that the user intended. The second problem is that the majority of the processing overhead is at query-time during computation of the minimum Steiner tree or the enumeration of all paths to find the minimum cost path.

One reason why query inference systems have not been widely deployed is that they are tied to query language implementations, which limits the types of queries allowed. Each approach to query inference has been directly linked to a specific query language and hard-coded into the mapping algorithm that converts the query into SQL. These solutions are inherently not configurable. A query inference system infers natural joins, but can handle complex queries with theta joins and subqueries if the query language supports them.

In summary, none of the approaches is suitable in general. Although maximal objects are relatively efficient to process at query time, they are expensive to generate, may be semantically redundant, grow rapidly as the schema increases in size, and cannot support queries with complex joins. The cost-based approaches generate limited query interpretation, perform expensive computations at query-time, and cannot generate all valid interpretations for use in interactive query formulation.

3. AutoJoin Architecture

An ideal query inference system would automatically apply to existing relational schemas without administrator intervention, quickly pre-compute the necessary data structures to minimize overhead during query execution, and support interactive query formulation with the user by generating and explaining possible query interpretations. The goal is to separate query inference (which applies at the schema level not at the query interface level), so that it can be used with any query interface. Unlike previous query inference systems, AutoJoin *enumerates* interpretations without selecting a specific one. This allows the user of the query interface full control on how ambiguous queries are handled.

The AutoJoin architecture consists of a pre-processing phase and a query-time inference engine. The pre-processing step is performed only once, but may have to be re-performed if schema evolution occurs.[2] In this step, the schema information is extracted using standard API calls, transformed into a generic schema representation called a join graph, and optionally annotated to improve names and reduce ambiguity. Then, the system computes the maximal objects and stores all information including the maximal objects, join graph, annotation, and schema into an XML document to be used at query-time. Efficient algorithms are required for large schemas and for schemas that change frequently.

At query-time (see Figure 4), the user interacts with a query interface to build their query. The query interface may be a text language like attribute-only SQL or may be a graphical query tool using ER diagrams or QBE. The query interface allows the user to enter potentially ambiguous queries and then translates the user's query into a set of nodes and edges on the join graph. Nodes represent relations and edges represent joins. The query interface may provide configuration parameters such as its willingness to accept lossy joins and a ranking protocol to rank interpretations if the query is ambiguous. The `QueryBuilder` takes this information and uses the pre-computed information

2 Re-computation is necessary if schema evolution occurs that would modify the join graph such as adding/removing a relation (node) or a foreign key (edge).

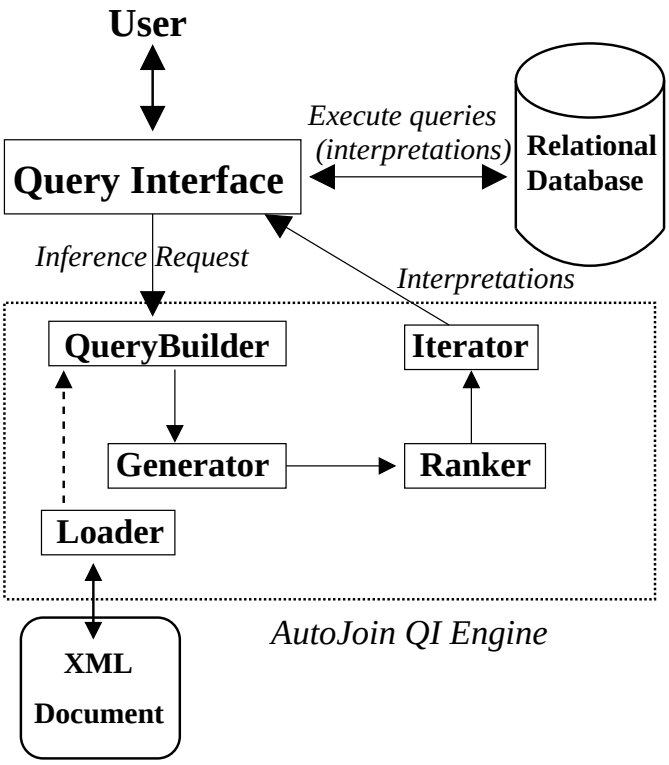

Figure 4. AutoJoin Architecture

stored in the XML document (which must be loaded once at startup for all queries) and calls the `Generator` to enumerate the potential interpretations. The `Generator` constructs all interpretations and passes them to the `Ranker` that uses any supplied ranking function to order the interpretations. Finally, the query interface uses the `Iterator` to return the interpretations in rank order. The query interface can then execute one or more interpretations on the database, or if the query is ambiguous, communicate the ambiguity to the user as necessary. Since the architecture only requires the query interface to specify the nodes of interest, the query interface has complete control over the inference process. The key components of the architecture are discussed in the following sections.

4. Representing Joins in Schemas

The potential joins in a relational schema are represented by a directed graph called a join graph.

Definition 1 *A join graph $JG = (N, E)$ for relational database schema S is a directed graph where:*

- *Each relational schema $R_i \in S$ is represented as a node $n_i \in N$.*
- *There is a directed edge $e = (n_i, n_j) \in E$ from node n_i (relation R_i) to node n_j (relation R_j) if there exists*

a foreign key constraint of the form $R_i[A] \subseteq R_j[B]$ where A and B are subsets of the attributes of R_i and R_j respectively.

part(p_partkey, p_name, p_mfgr, p_brand, p_type, p_size, p_container, p_retailprice, p_comment)
supplier(s_suppkey, s_name, s_address, s_nationkey, s_phone, s_acctbal, s_comment)
partsupp(ps_partkey,ps_suppkey,ps_availqty,ps_supplycost,ps_comment)
customer(c_custkey,c_name,c_address,c_nationkey,c_phone,c_acctbal, c_mktsegment,c_comment)
orders(o_orderkey, o_custkey, o_orderstatus, o_totalprice, o_orderdate, o_orderpriority, o_clerk, o_shippriority, o_comment)
lineitem(l_orderkey, l_partkey, l_suppkey, l_linenumber, l_quantity, l_tax, l_extendedprice, l_discount, l_returnflag, l_linestatus, l_shipdate, l_commitdate, l_receiptdate, l_shipinstruct, l_shipmode, l_comment)
nation(n_nationkey, n_name, n_regionkey, n_comment)
region(r_regionkey, r_name, r_comment)

lineitem(l_partkey) $\subseteq$ partsupp(ps_partkey) $\subseteq$ part(p_partkey)
lineitem(l_suppkey) $\subseteq$ partsupp(ps_suppkey) $\subseteq$ supplier(s_suppkey)
lineitem(l_partkey,l_suppkey) $\subseteq$ partsupp(ps_partkey,ps_suppkey)
orders(o_custkey) $\subseteq$ customer(c_custkey)
customer(c_nationkey) $\subseteq$ nation(n_nationkey)
supplier(s_nationkey) $\subseteq$ nation(n_nationkey)
lineitem(l_orderkey) $\subseteq$ orders(o_orderkey)
nation(n_regionkey) $\subseteq$ region(r_regionkey)

Figure 5. TPC-H Schema

A join graph can be automatically built from an existing relational schema by extracting relation names and foreign key constraints. A special case is handling 1:1 relationships. A 1:1 relationship can be detected in the schema by finding a foreign key that must be unique. The general solution is to add an inverse edge $e^{-1} = (v, u)$ for each 1:1 foreign key edge $e = (u, v)$. Inverse edges are recorded as they affect the maximal object generation algorithm. In certain cases, inverse edges are not used if the foreign key is not-null and unique (for instance the foreign key is also the primary key of the relation). Then, the two nodes u and v are merged into a single node uv. The example TPC-H schema used throughout this paper is in Figure 5, and its associated join graph is in Figure 6.

If the schema does not explicitly represent foreign keys, they can be added to the join graph using annotation. This allows joins to be added that cannot be automatically detected. Some lossy joins may not be automatically extracted if the schema does not have two or more foreign keys to the same relation. For instance, if the *Nation* relation did not exist in TPC-H, there still would be a lossy join between *Customer* and *Supplier* on nation name, although that could not be detected using only foreign keys. In this case, a dummy node *Nation* would be added with two incoming edges to represent the lossy join. Finally, it is possible to have multiple edges between two nodes if there are two or more foreign keys between them. In this case, edges are distinguished by labeling them with the foreign key attributes.

A join graph is similar to a graph of inclusion dependencies, which have been used in other domains [15, 16]. However, join (inclusion dependency) graphs have not been previously used for query inference. Previous query inference systems used hypergraphs [18] or undirected ER graphs [22, 23], both of which can be modeled by join graphs. A hypergraph in [18] has a node for each attribute and an edge connecting all attributes for each relation. This representation can be converted into a join graph by mapping each hypergraph edge to a join graph node, and constructing an edge between two nodes in the join graph if the corresponding two edges in the hypergraph share one or more nodes (attributes). The direction of the edge can be established using the given functional dependencies. Translating an ER graph into a join graph is similar to the algorithm for translating an ER model into a relational model. Entities in the ER graph are nodes in the join graph. A relationship is converted to a node with outgoing edges to its participating entities if it is n-ary or it is binary with a cardinality of M:N. A 1:N relationship is converted into a directed edge then from the N-side node to the 1-side node. If the relationship is 1:1, an edge and its inverse is added to the join graph.

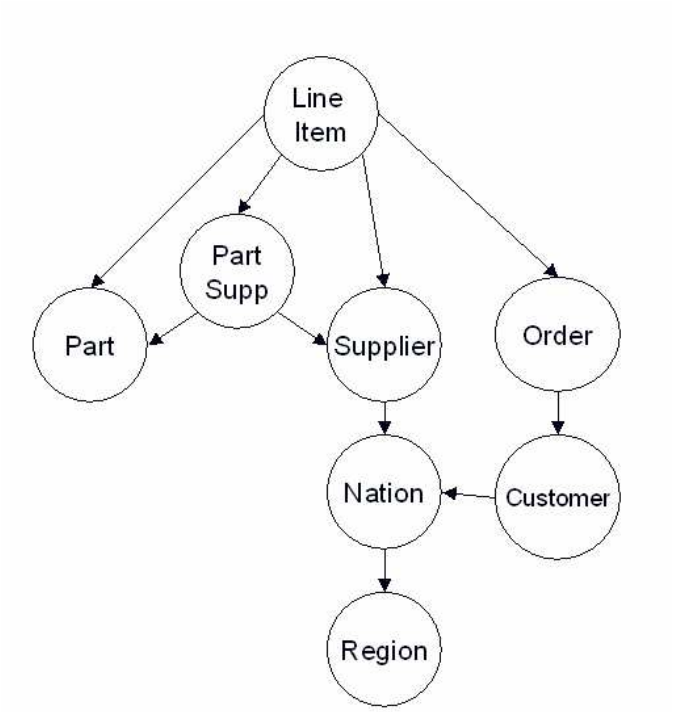

Figure 6. Join Graph for TPC-H Database

Using a join graph, the maximal object approach reduces to the problem of finding all connected, maximal subtrees of the join graph. The cost-based approach reduces to the problem of finding the minimum directed Steiner tree of the join graph. If only lossless interpretations are required, all edges are assigned a cost of one. If lossy interpretations are allowed, add to the join graph reverse edges with high con-

stant cost. Finally, the test for ambiguity is changed when using a directed graph. In an undirected graph, a schema is ambiguous if it contains an undirected cycle. In a join graph, a schema is *ambiguous* if it contains a directed cycle or if it contains at least one node with two or more incoming edges.

We now formally define the meaning of a query, query interpretation, and query inference.

Definition 2 *A user* query $Q = (N', E')$ *on a join graph* $JG = (N, E)$ *is a subgraph of* JG *such that* $N' \subseteq N$ *and* $E' \subseteq E$.

For use in query inference, a user query reduces to a set of specified nodes (relations) and edges (natural joins). A node is specified if one or more of its attributes are required for a selection, projection, ordering, or grouping operation. A specified edge is a natural join condition explicitly given by the user. The query interface must translate the user query into the required form.

Definition 3 *A* query interpretation $QI = (N', E')$ *on a join graph* $JG = (N, E)$ *is a* **connected** *subgraph of* JG *such that* $N' \subseteq N$ *and* $E' \subseteq E$.

A query interpretation is usually a tree, but may be a graph. We use the terms *join tree* and *join graph* for query interpretations that are trees and graphs respectively.

Definition 4 *The* query inference problem *requires enumerating and ranking query interpretations of the user query such that the query interpretation desired by the user is among the highest ranked interpretations.*

Previously, query inference was defined as selecting a single interpretation for the user. This is impractical because a system will never be able to always select the correct interpretation. We re-formulate the query inference problem similar to an information retrieval problem. The goal of query inference is to enumerate potential query interpretations and present them to the user in order of the likelihood that the interpretation is what the user intends. This is similar to how a web search engine ranks pages according to the expected value for the user. Query inference then becomes two related problems. First, given a subset of nodes and edges of the join graph, the system must enumerate the possible connections between them. Then, it should rank interpretations in order of expected value.

5. Generating Maximal Objects Efficiently

Previous approaches for generating maximal objects started with a node [17] or an edge [22] in the graph and expanded using lossless joins in all possible ways. The growing algorithm was inefficient, and often non-computable, for large graphs. Our approach, called EMO (Enumerate Maximal Objects), constructs the maximal objects without expanding in all possible ways. The basic idea is since maximal objects are trees, a root of a maximal object cannot be a node with an incoming edge unless that node is in a strongly connected component (SCC). The algorithm identifies all nodes that have only outgoing edges or are in a strongly connected component with no incoming edges from outside the component. Only those nodes can be roots of maximal objects. From each root, the reachable subgraph is found by traversing directed edges and an algorithm for calculating all spanning trees of the reachable graph is used [11] to find all maximal objects for each subgraph. Repeating the process for every potential maximal object root produces all maximal objects without constructing any duplicates. The algorithm for enumerating all spanning trees is from [11] and has performance $O(V + E + EN)$, where N is the number of spanning trees of the graph. If the join graph has inverse edges, a final step is used to remove equivalent maximal objects that are identical except that one contains an edge and the other its inverse. The algorithm is in Figure 7. The algorithm can be simplified if the graph is known not to have any SCCs in which case it is sufficient to run `findReachable()` only on nodes of the graph with no incoming edges.

```
EMO (JoinGraph dg, List maxObjs)
{
    allSCC = Find all strongly connected components of dg
    rGraphs = ∅
    for each scc in allSCC
        if (nodes of scc == 1 and node n of scc has no in-edges)
            rGraphs = rGraphs ∪ findReachable(n);
        else
        {
            for each node n in scc
                if (n has no incoming edges from outside scc)
                    rGraphs = rGraphs ∪ findReachable(n);
        }
    maxObjs = ∅
    for each reachable graph g in rGraphs
        maxObjs = maxObjs ∪ g.findSpanningTrees()
    maxObjs = removeInverseObjects(maxObjs)
}
```

Figure 7. EMO Algorithm

As the number of spanning trees may be exponential (such as for a completely connected graph), the performance of the algorithm in the worst-case may be exponential. In practice, database graphs do not exhibit the worst case and tend to have a reasonable number of maximal objects. EMO outperforms a grow all ways approach both in

time and number of edges and nodes visited. The improved performance is because EMO does not duplicate effort and grows only from valid maximal object roots. The eight maximal objects for the TPC-H database are in Figure 8.

The following two proofs demonstrate the correctness of the algorithm by showing that EMO generates all maximal objects and only valid maximal objects.

Theorem 1 *EMO produces all maximal objects.*

Proof: Proof by contradiction. Let A be a maximal object that is not produced by EMO. Let r be the root of A. Let C be the strongly connected component containing r.
Case 1: If C contains only one element, then r must have no incoming edges. Otherwise, suppose it has incoming edge (x, r). Adding (x, r) to A will produce a bigger tree containing A. That contradicts the fact that A is a maximal object. So EMO will find the reachable graph of r.
Case 2: If C contains more than one element, then EMO finds the reachable graph of every node in C with no incoming edges from outside C. If r has no incoming edges from outside C, EMO will find the reachable graph of r. If r has incoming edges from outside C, then similar to Case 1, A is not a maximal object.
In either case, A is one of the spanning trees of the reachable graph of r. Thus, A must be produced by EMO.

Theorem 2 *Any spanning tree produced by EMO is a maximal object.*

Proof: Proof by contradiction. Suppose A is a tree produced by EMO, but A is not a maximal object. Then A must be a subtree of some maximal object B. By Theorem 1, EMO produces B.
Case 1: Suppose A and B have the same root r. Then A and B are both spanning trees of the reachable graph of r. This contradicts the fact that A is a smaller tree contained in B.
Case 2: Suppose the root rA of A is contained in B but is not the root of B. Then rA must have an incoming edge $e = (x, rA)$ where x is not contained in A. EMO would not have found the reachable graph for rA if it had an incoming edge unless rA was in a strongly connected component C. Edge e may be an incoming edge from outside C or an edge within C. If it is an edge in C, x must be in A since x would be reachable from rA. A contradiction as x is not in A. If e is an incoming edge from outside C, then EMO will not produce a spanning tree of the reachable graph from rA. Thus, A would never be produced.

6. Eliminating Equivalent Maximal Objects

The number of maximal objects is dictated by the ambiguity inherent in the schema. However, semantically equivalent maximal objects may exist that have distinct subgraphs. This is especially common in hierarchically structured databases where the primary key of one relation contains the primary key of its parent relation. Consider an example schema of $R(\underline{A}, G), S(\underline{A}, \underline{B}, H)$ and $T(\underline{A}, \underline{B}, \underline{C}, I)$ where $T(A) \subseteq S(A) \subseteq R(A)$, $T(A, B) \subseteq S(A, B)$. This schema has two distinct maximal objects, but there is actually only one semantic interpretation. It is critical that these equivalent maximal objects be detected and only one semantic interpretation preserved, otherwise queries may be incorrectly determined to be ambiguous.

Semantically equivalent maximal objects will have subgraphs that contain the same nodes but may have one or more different edges connecting the nodes. A *shortcut join* exists when there is a join between two relations that is semantically equivalent to a longer join path of two or more edges. We denote a natural join on foreign key attributes X between two relations R_i and R_j as $R_i \bowtie_X R_j$. To simplify the discussion, the set of attributes X is assumed to have the same name in both relations, although in practice this is not required.

Definition 5 *A* shortcut join *between two relations R_i and R_j is a natural join on attributes X where $X \subseteq R_i$ and $X \subseteq R_j$ and there exists a join path $R_i \bowtie_{X_1} T_1 \bowtie_{X_2} T_2 \bowtie_{X_3} ... \bowtie_{X_n} T_n \bowtie_Z R_j$ where $Z = X$ and $Z \subseteq X_n \subseteq X_{n-1} ... \subseteq X_1$.*

Using the previous example schema, $T \bowtie_A R$ is a shortcut join for the longer join path $T \bowtie_{A,B} S \bowtie_A R$.

A shortcut join is equivalent to the longer join path based on functional dependencies. The functional dependency $R_i[X] \rightarrow R_j$ of the shortcut join is equivalent to $R_i[Y] \rightarrow T[Z] \rightarrow R_j$ since $X \subseteq Y$ and $Z = X$. Note that it is insufficient for only $Z = X$ as the relation R_j may join with T on X but assign a different role to X than R_i. For example, assume the *LineItem* relation in TPC-H had a foreign key to *Customer* indicating the customer that the particular item should be shipped to. Also, assume that the foreign key in the *Order* relation to *Customer* refers to the customer who pays for the order (BillTo). In this case, the join between *LineItem* and *Order* is not a shortcut join for the longer join path *LineItem*, *Order*, *Customer* as the customer serves two different roles (ShipTo and BillTo).

Shortcut joins can be detected while building the join graph. If a relation R_i has two foreign keys on sets of attributes Y and X to relations R_j and R_k respectively, where $X \subset Y$ then the join from R_i to R_k on X is a shortcut join and is not added to the join graph. Shortcut joins are maintained in a list and are re-inserted at query-time when the shortcut join can be used to avoid a longer join path.

There are two shortcut joins in the TPC-H schema: *LineItem* to *Part* and *LineItem* to *Supplier*. By removing these joins, the number of maximal objects is reduced from 8 to 2 (maximal objects 1 and 5), and the number of unam-

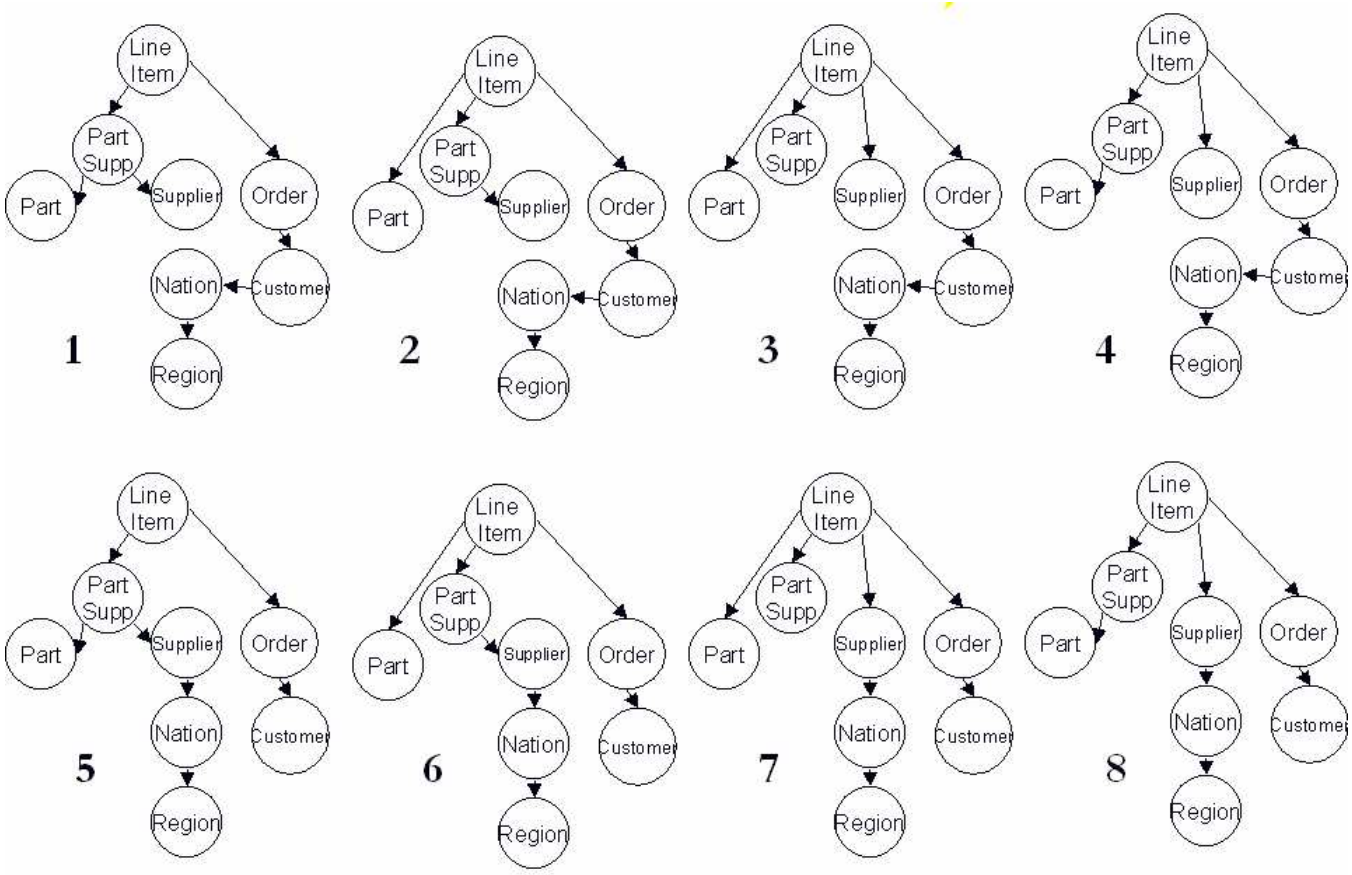

Figure 8. Maximal Objects for TPC-H Database

biguous queries increases to 26% from 8%. At query-time, if a join specifies only *LineItem* and *Part*, the path in the maximal object will be: *LineItem-PartSupp-Part* which is replaced by the shortcut join *LineItem-Part*.

7. Query-Time Processing

To minimize processing time, several performance improvements are implemented. First, identifying potential maximal objects is performed using a reverse index where each relation (node) has a list of its corresponding maximal objects. The index entry of each requested node is found using hashing. The system takes the intersection of all index entries to determine the set of maximal objects that contain all the nodes. The second improvement is an efficient method for building minimal query interpretations. Instead of recursively pruning leaf nodes of the maximal object, a minimal join tree is built by performing a union of the paths from the least common ancestor of all requested nodes to each requested node. This approach is more efficient when the maximal objects are large as the least common ancestor information can be precomputed and reused.

Our algorithm for generating query interpretations is:

- Identify the maximal objects that contain all the user specified nodes (and edges).
- Build minimal query interpretations from the maximal objects such that all leaf nodes are specified nodes.
- Eliminate any duplicate query interpretations.

This algorithm will generate all possible lossless interpretations of the user query. Queries with attributes not all contained in at least one maximal object cannot be answered using this approach.

Consider the sample query in Figure 3 that relates a part name with a nation name. One possible interpretation, derived from maximal object 1, is return all parts ordered by customers in the United States. This interpretation corresponds to the SQL query in Figure 1. In the join graph with no shortcut edges removed, all eight maximal objects contain the nodes *Part* and *Nation* required by the query. There are six distinct interpretations generated as maximal objects 1 and 4 result in the same interpretation after pruning. Maximal objects 2 and 3 are also identical after pruning. The last four maximal objects 5 to 8 all produce distinct queries. However, there are actually only two semantically distinct interpretations: nation can relate either to customer or to supplier. When shortcut edges are removed from the join

graph, only these two interpretations are generated. This is an example of an ambiguous query where the query interface must either execute and union the results of both interpretations, select an interpretation automatically, or ask the user for assistance.

In the AutoJoin architecture (Figure 4), the `Generator` module implements the algorithm to generate all possible lossless interpretations and provides them to the `Ranker`. A default ranking system is based on the number of edges (joins) in the interpretation. The query interface uses the `Iterator` to process the interpretations in rank order.

8. Handling Lossy Joins

The maximal object approach only generates queries with lossless joins. Although lossy joins are rare, there are some queries that use them, such as query 5 in TPC-H. There are an infinite number of interpretations with lossy theta joins. The algorithm presented generates all minimal, one lossy join interpretations where the lossy join is a equijoin connecting foreign keys.[3] Lossy joins on shared attribute domains are studied because they are the only ones that can be reasonably inferred from the schema. Other theta joins must be specified by the user (see Section 9.1). Since there can be a large number of such interpretations and they will be requested rarely, they are computed at query-time using the existing maximal objects.

Definition 6 *A* minimal one-lossy join interpretation *on foreign keys (shared attribute domain) is a minimal connected subgraph of the join graph that contains all user specified nodes and edges and one lossy equijoin connecting tables with a shared attribute domain such that removal of any node or edge would result in a graph that is disconnected or does not contain all specified nodes and edges.*

Definition 7 *Let A and B be trees that represent two maximal objects. Let $UAB = A \cup B$. A* lossy node *of UAB is any node in UAB with two incoming edges.*

The general idea is that all one lossy join interpretations can be found by combining pairs of maximal objects that together contain all the requested nodes. The union of this pair of maximal objects will contain one or more lossy nodes. Since only one lossy node is required, the algorithm must pick one lossy node, delete one incoming edge for all the other lossy nodes in all possible combinations, and then validate and prune the resulting graph. The algorithm in Figure 9 enumerates all such graphs in a breadth-first manner one lossy node at a time. The union of two maximal objects (trees) may result in a graph with cycles. If the graph has an undirected cycle (by treating directed edges as undirected), multiple interpretations are generated by removing each edge in turn from the cycle. The prune method recursively performs these steps:

- If a node has no outgoing edges, one incoming edge, and it is not a target node, delete the node and the incoming edge.
- If a node has no incoming edge, only one outgoing edge, and it is not a target node, delete it and the outgoing edge.

A proof of correctness of the algorithm is omitted due to space. It is clear that the algorithm will produce only valid minimal one lossy join interpretations as the `validate()` method checks each interpretation generated. It can be shown that it produces all such interpretations as the algorithm always generates a set of graphs that are supergraphs of any valid interpretation and the smallest of these supergraphs is always the minimal interpretation desired.

9. Handling Ambiguity

An ambiguous query has multiple interpretations. A single interpretation may be automatically selected based on user parameters or iteratively selected by interaction with the query inference engine. Our inference engine will enumerate and rank all lossless query interpretations. A unique interpretation occurs if there exists a unique join tree that contains the desired attributes. This is quite common, especially for data warehouses with star schemas. A query may be unambiguous even if the schema is ambiguous. If more than one interpretation exists, then the union of the interpretations is commonly used, but it is also possible to pick an interpretation using shortest path, lowest cost, or some other metric. Query inference systems can communicate the inferred query back to the user [20, 24] using Pseudo-English explanations generated from a schema annotated with meaningful names for the entities and relationships. Thus, ambiguous queries can be refined to unambiguous queries with user interaction.

There are two common sources of ambiguity in schemas:

- A single relation storing an entity that plays multiple roles results in a node in the join graph with two or more incoming edges.
- Multiple semantic relationships between entities result in directed cycles and strongly connected components in the join graph.

The first source of ambiguity is common as schemas have shared lookup tables used by multiple relations. Examples include relations for addresses, people, companies, and status codes. An example in TPC-H is *Nation*

3 Foreign keys are used here to detect a shared attribute domain (such as Nation for Supplier and Customer in TPC-H). As mentioned in Section 4, such shared domains are included in the join graph by manual annotation if they cannot be detected using foreign keys.

```
SingleLossyJoins (List maxObjs, List queryNodes)
{   joinGraphs = ∅
    for each pair (A,B) of maximal objects {
        UAB = A ∪ B
        if UAB contains all queryNodes {
            L = all lossy nodes of (A,B)
            JGAB = ∅ // All join graphs for (A,B)
            for each node n in L {
                levelGraphs = { UAB }
                for each node m in L - {n} {
                    nextLevelGraphs = ∅
                    for each graph g in levelGraphs {
                        JG1 = delete one in-edge of m in g
                        JG2 = delete other in-edge of m in g
                        Add JG1 and JG2 to nextLevelGraphs
                    }
                    levelGraphs = nextLevelGraphs
                }
                JGAB = JGAB ∪ validatePrune(levelGraphs)
            }
            joinGraphs = joinGraphs ∪ JGAB
        }
    }
}

validatePrune (List graphs)
{   result = ∅
    for each graph G in graphs
        result = result ∪ doCycle(G)
    return validate(result)
}

doCycle (Graph G)
{   result = ∅
    if G is acyclic
        result = Prune(G)
    else if G has an undirected cycle
        for each edge e in cycle {
            G' = Prune(G - e)
            result = result ∪ doCycle(G')
        }
    return result
}
```

Figure 9. One Lossy Join Algorithm

which serves the two semantic roles of storing the nation of customers and the nation of suppliers. In another example schema (*Tenant*), there were 36 foreign keys (directed edges) from one table to another table that stored status codes. This form of ambiguity arises when distinct attributes have the same underlying domain of values. Attribute renaming [23, 24] resolves this ambiguity by encoding a role name as part of the attribute name. Thus, a user in TPC-H may query for *Customer.Nation.Name* or *Supplier.Nation.Name* instead of *Nation.Name*. Introducing these two names makes *all* queries on TPC-H unambiguous. Ambiguity is removed as the role name selects both a node and an edge in the join graph.

An example of the second source of ambiguity is a database storing employees and departments where an employee has a department and a department has a manager (which is also an employee). These two relationships result in a directed cycle between employee and department in the join graph. Once again, the ambiguity can be reduced by querying on role names. The query `SELECT Employee.Name, Department.Name` is ambiguous, but the query `SELECT Manager.Name, Department.Name` is not.

Overall, ambiguity can be reduced by introducing new attributes that encode role semantics in their names (and implicitly select the appropriate join required). More work is necessary in this area especially in automatically generating these role names. Although the number of maximal objects are not decreased by using role names, the number of ambiguous queries is greatly reduced.

9.1. Beyond Natural Joins

The focus of this work is on natural joins on foreign keys as they are the most common type of join. However, query inference is not restricted to queries with only natural joins. In general, any complex join condition can be specified by the user, and the query inference engine will infer any natural joins still required to complete the query.

User specified joins are divided into two types. A user may specify a natural join using a foreign key in which case the join is treated as a specified edge and all valid interpretations (maximal objects) must contain all the specified nodes *and edges*. A theta join is handled by merging the two join nodes in the join graph and updating the edges. Due to EMO's efficiency, it is possible to re-compute the maximal objects for the modified join graph and apply the regular query inference algorithm. Increased performance is possible by updating the graphs of only the existing maximal objects that apply to the query. Query inference can be applied to any query including ones that use subqueries, complex join predicates, and outer joins. Maier and Ullman [17] showed that the maximal object approach can be

used with queries with tuple variables. Tuple variables are required when one or more relations occur in the query multiple times. The query interface must allow the user to specify such complex queries in a convenient fashion, but the presence of such complexity does not limit the usefulness of query inference.

10. Performance Experiments

The AutoJoin query inference engine is implemented in Java and uses the JDBC API to extract schema information. Experiments involving query execution do not include parse time as the SQL query is converted into the join graph representation before inference begins. All schemas are in production use at various organizations or extracted from the Internet. The largest schema is *caBIO* from the NCI Cancer Grid project (caBIG)[4] which contains 149 nodes, 213 edges, and 1253 maximal objects. The experiments were performed on a 1.3 GHz AMD Athlon with 512 MB of memory running Windows XP.

The first experiment compares the performance of EMO with the grow all ways approach used in [17, 22]. The results in Figures 10 and 11 show that EMO significantly outperforms the grow all ways approach both in terms of time and edge visits. The grow all ways approach is very inefficient and cannot complete *caBIO* without running out of memory.[5] Both approaches are comparable time-wise for very small graphs. EMO outperforms on large graphs by an order of magnitude, which is especially important when schema evolution is frequent. *EMO is able to handle schemas that could not even be processed by the current approach.* EMO's absolute performance is sufficiently fast to support generation of maximal objects at query-time for almost all graphs.

The second experiment determines how removing shortcut joins reduces ambiguity. We use the TPC-H schema and determine the percentage of queries that are unambiguous with and without removing shortcut joins. All 255 potential queries were treated as equally likely (all queries of one table, of two tables, etc.). Removing shortcut joins improved the number of unambiguous queries from 8% to 26% (see Figure 12). Of the 22 benchmark TPC-H queries, removing shortcut joins made 68% of the queries unambiguous versus 45% originally. Several queries in TPC-H contain nested subqueries. We consider a query to be unambiguous

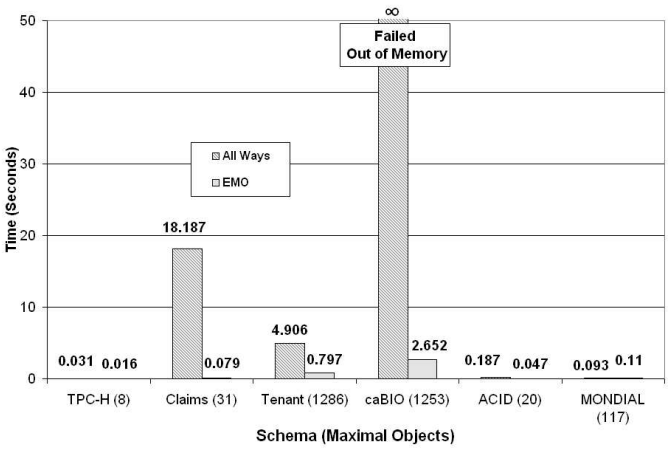

Figure 10. Time to Compute Maximal Objects

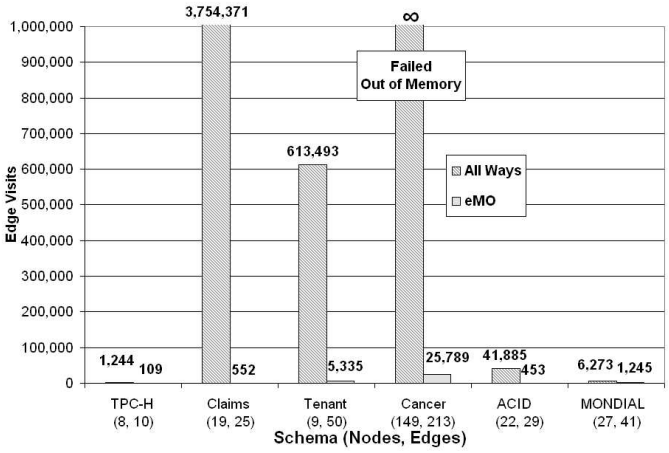

Figure 11. Edge Visits to Compute Maximal Objects

if both the join tree for the subquery and outer query can be inferred. Query 5 contains a lossy join, and two queries (7 and 8) require two copies of *Nation* that must be specified by the user. The hierarchical *EDS* schema shows an even greater improvement by removing shortcut joins.

The overhead of performing query inference for each query must be minimal. In a third experiment, we determine the time to perform query inference on various schemas. We calculated the average time to infer the joins for the 22 benchmark TPC-H queries, for all possible 255 TPC-H queries, and for two table queries in the sample databases. As Figure 13 shows, the average query inference time is well below 10 ms even for large schemas. Removing shortcut joins improves the time for TPC-H. The inference time for *caBIO* increases slightly despite the significantly larger

4 http://cabig.nci.nih.gov/

5 The grow all ways algorithm is either CPU or memory constrained depending on its implementation. If growth is in a BFS manner, a major effi ciency improvement occurs by eliminating duplicate graphs at each level of the tree, although this requires a huge amount of memory as there are often millions of intermediate graphs. If duplicates are not eliminated, a DFS approach can be used which results in running times 100-1000 times slower (BFS times are shown). The DFS approach did not complete for the *Claims* database.

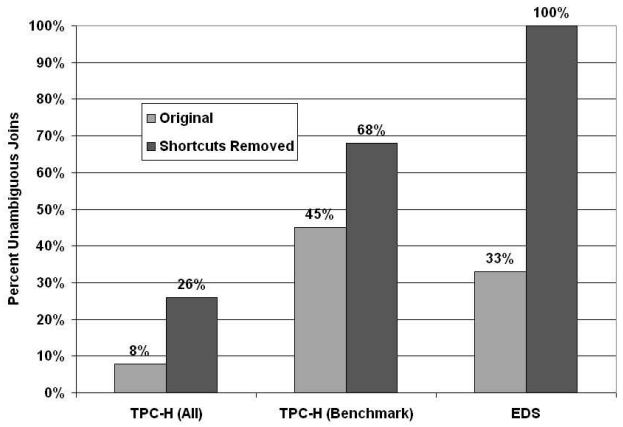

Figure 12. Reducing Ambiguity by Removing Shortcut Joins

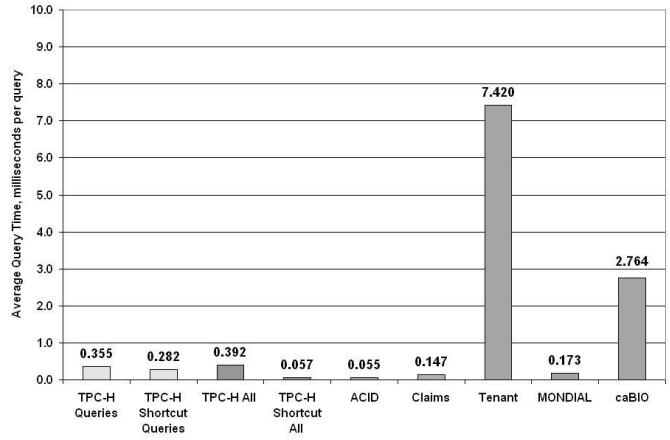

Figure 13. Query Inference Time

schema size. By building query interpretations using least common ancestor versus recursive pruning, the average query time decreased from 126 ms to 2.7 ms. The lossy join in TPC-H query 5 is specified for this experiment. If it was not specified, the time to compute all lossless and one-lossy interpretations (using the algorithm in Section 8) was 16 ms (after removing shortcut joins). Thus, query inference time is minimal both in absolute terms and relative to the time to parse and execute the query.

11. Conclusion

Our contribution is demonstrating how query inference can be efficiently performed on large schemas by providing scalable algorithms for enumerating and processing query interpretations. The enumeration algorithm, EMO, generates all semantically distinct interpretations significantly faster than previous approaches. Another algorithm was provided that enumerates all one lossy interpretations, which was not previously possible. Query inference is generalized to support complex queries that have user-specified joins, and the approach is usable with any relational schema and query interface. Our prototype query inference engine called AutoJoin allows the user to configure how the query interpretations are ranked. Several performance experiments demonstrate that the overhead of query inference is minimal, and show how ambiguity can be reduced by removing shortcut joins. Overall, this paper establishes that query inference is a practical query tool that should be incorporated into query interfaces and database systems.

Future work includes detailed case studies on the structure of relational schemas and sources of ambiguity. These studies will be used to devise new methods for handling ambiguity and to create a query interface that utilizes the query inference engine to improve user querying. Ranking strategies will be evaluated on their effectiveness in highly ranking the user's intended query interpretation.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer:A System for Keyword-Based Search Over Relational Databases. In *IEEE ICDE*, pages 5–16, 2002.

[2] I. Androutsopoulos, G. Ritchie, and P. Thanisch. MASQUE/SQL - An Effi cient and Portable Natural Language Query Interface for Relational Databases. Technical report, University of Edinburgh, 1998.

[3] M. Angelaccio, T. Catarci, and G. Santucci. QBD*: A Graphical Query Language with Recursion. *IEEE Trans. Software Eng.*, 16(10):1150–1163, 1990.

[4] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank:Authority-Based Keyword Search in Databases. In *VLDB*, 2004.

[5] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the Desirability of Acyclic Database Schemes. *J. ACM*, 30(3):479–513, 1983.

[6] F. Benzi, D. Maio, and S. Rizzi. Visionary: A Visual Query Language Based on the User Viewpoint Approach. In *Interfaces to Databases*, 1996.

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

[8] T. Catarci. What happend when Database Researchers met Usability. *Information Systems*, 25(3):177–212, 2000.

[9] T. Catarci, M. Costabile, S. Levialdi, and C. Batini. Visual Query Systems for Databases: A Survey. *Journal of Visual Language Computing*, 8(2):215–260, 1997.

[10] B. Czejdo, R. Elmasri, M. Rusinkiewicz, and D. Embley. A Graphical Data Manipulation Language for an Extended Entity-Relationship Model. *IEEE Computer*, 23(3):26–36, 1990.

[11] H. Gabow and E. Myers. Finding All Spanning Trees of Directed and Undirected Graphs. *SIAM Journal of Computing*, 7(3):280–287, 1978.

[12] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. In *VLDB*, 1998.

[13] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. In *VLDB*, 2003.

[14] V. Hristidis and Y. Papakonstantinou. DISCOVER:Keyword Search in Relational Datbases. In *VLDB*, 2002.

[15] D. Lee, M. Mani, F. Chiu, and W. Chu. NeT & CoT: Translating relational schemas to XML schemas using semantic constraints. In *Proceedings of the 11th CIKM*, pages 282–291, 2002.

[16] M. Levene and G. Loizou. Guaranteeing no interaction between functional dependencies and tree-like inclusion dependencies. *Theor. Comput. Sci.*, 254(1-2):683–690, 2001.

[17] D. Maier and J. Ullman. Maximal Objects and the Semantics of Universal Relation Databases. *TODS*, 8(1):1–14, 1983.

[18] D. Maier, M. Vardi, and J. Ullman. On the Foundations of the Universal Relation Model. *ACM Transactions on Database Systems*, 9(2):283–308, June 1984.

[19] A. Motro. Constructing queries from tokens. *SIGMOD Record*, 15(2):120–131, June 1986.

[20] V. Owei and S. Navathe. Enriching the conceptual basis for query formulation through relationship semantics in databases. *Information Systems*, 26(6):445–475, 2001.

[21] A. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, 2003.

[22] R. Semmel and J. Mayfield. Automated Query Formulation using an Entity-Relationship Conceptual Schema. *Intelligent Information Systems*, 8:267–290, 1997.

[23] J. Wald and P. Sorenson. Resolving the Query Inference Problem Using Steiner Trees. *TODS*, 9(3):348–368, 1984.

[24] J. Wald and P. Sorenson. Explaining Ambiguity in a Formal Query Language. *TODS*, 15(2):125–161, 1990.

[25] X. Wu and T. Ichikawa. KDA: A Knowledge-Base Database Assistant with a Query Guiding Facility. *IEEE Trans. Knowl. Data Eng.*, 4(5):443–453, 1992.

[26] G. Zhang, F. Meng, G. Kong, and W. Chu. Query Formulation from High-leve Concepts for Relational Databases. In *User Interfaces to Data Intensive Systems*, 1999.

[27] Z.-Q. Zhang and A. O. Mendelzon. A Graphical Query Language for Entity-Relationship Databases. In *ER 1983*, pages 441–448, 1983.

[28] M. Zloof. Query-by-Example: the Invocation and Definition of Tables and Forms. In *VLDB*, pages 1–24, 1975.