DISCOVER: Keyword Search in Relational Databases*

Vagelis Hristidis
University of California, San Diego
vagelis@cs.ucsd.edu

Yannis Papakonstantinou
University of California, San Diego
yannis@cs.ucsd.edu

Abstract

DISCOVER operates on relational databases and facilitates information discovery on them by allowing its user to issue keyword queries without any knowledge of the database schema or of SQL. DISCOVER returns qualified joining networks of tuples, that is, sets of tuples that are associated because they join on their primary and foreign keys and collectively contain all the keywords of the query. DISCOVER proceeds in two steps. First the Candidate Network Generator generates all candidate networks of relations, that is, join expressions that generate the joining networks of tuples. Then the Plan Generator builds plans for the efficient evaluation of the set of candidate networks, exploiting the opportunities to reuse common subexpressions of the candidate networks.

We prove that DISCOVER finds without redundancy all relevant candidate networks, whose size can be data bound, by exploiting the structure of the schema. We prove that the selection of the optimal execution plan (way to reuse common subexpressions) is NP-complete. We provide a greedy algorithm and we show that it provides near-optimal plan execution time cost. Our experimentation also provides hints on tuning the greedy algorithm.

1 Introduction

Keyword search is the most popular information discovery method because the user does not need to know either a query language or the underlying structure of the data. The search engines available today provide keyword search on top of sets of documents. When a set of keywords is provided by the user, the search engine returns all documents that are associated with these keywords. Typically, two keywords and a document are associated when the keywords are contained in the document and their degree of associativity is often their distance from each other.

In addition to documents, a huge amount of information is stored in relational databases, but information discovery on relational databases is not well supported. The user of a relational database needs to know the schema of the database, SQL or some QBE-like interface, and the roles of the various entities and terms used in the query. The user of DISCOVER does not need knowledge of any of the above. Instead, DISCOVER enables information discovery by providing a straightforward keyword search interface to the database.

For example, consider the TPC-H schema shown in Figure 1 and the instance in Figure 2. The arrows in Figure 1 point in the direction of the primary to foreign key (one-to-many) relationships between tables. Consider a user searching for information on the association of the keywords "Smith" and "Miller". DISCOVER provides a simple interface where the user simply types the keywords – as he would do on a search engine. According to DISCOVER, an association exists between two keywords if they are contained in two associated tuples, i.e., two tuples that join through foreign key to primary key relationships, which potentially involve more tuples. This form of association is particularly useful and challenging. (We comment on other association criteria at the end.) DISCOVER does not require from the user to know the relations and the attributes where the keywords are found.

The solution to the query are the two *minimal joining sequences* that contain the keywords "Smith" and "Miller", namely $o_1 \bowtie c_1 \bowtie o_2$ and $o_1 \bowtie c_1 \bowtie n_1 \bowtie c_2 \bowtie o_3$. They are minimal in the sense that no tuple can be excluded and still have a sequence that contains the keywords. We use the notation $a \bowtie b$ to denote that tuple a joins with tuple b on their primary key to foreign key relationship. The first *joining sequence* shows that both "Smith" and "Miller" are clerks that have served customer Brad Lou, whereas the second merely says that the clerks have served customers Brad Lou and George Walters respectively, who both come from the USA. Intuitively, the first joining sequence is more useful than the second because it shows a closer association

*Work supported by NSF Grant No. 9734548.

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

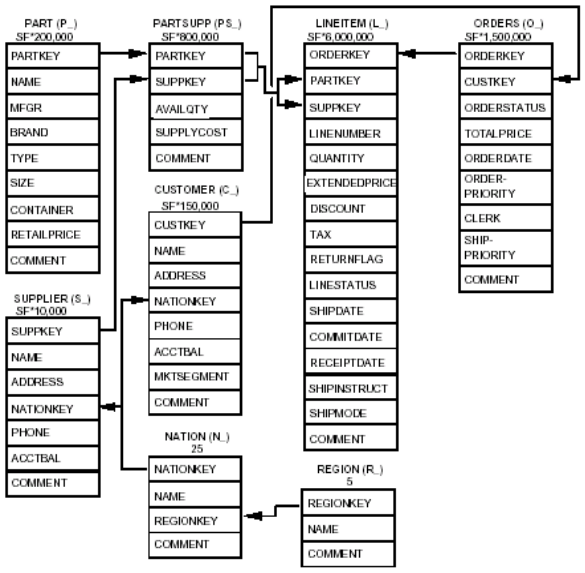


Figure 1: The TPC-H schema (copied from www.tpc.org)

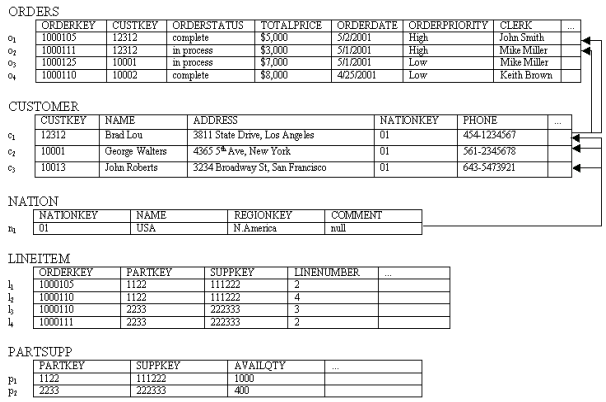


Figure 2: Sample TPC-H database instance

between "Smith" and "Miller". Based on the generalization of this intuition we rank join sequences according to the number of joins they involve. DISCOVER outputs the shorter sequences first.

When more than two keywords are involved, a minimal joining sequence may not be sufficient to represent a solution. Hence we introduce *minimal joining networks*, that are trees of tuples where any two adjacent tuples join through a primary key to foreign key relationship.

A high level representation of the architecture DISCOVER uses to find the joining networks is shown in Figure 3. First, the user gives a set of keywords $k_1, \ldots, k_m$ to the system. These keywords are looked up in the master index, which returns the *tuple sets* $R_i^{k_1}, \ldots, R_i^{k_m}$ for each relation R_i. Every tuple of $R_i^{k_j}$ contains keyword k_j as part of an attribute value. Then DISCOVER calculates all *candidate networks*, i.e., join expressions on foreign to primary key relationships of relations or tuple sets, as shown in Figure 3. The set of candidate networks is guaranteed to produce all

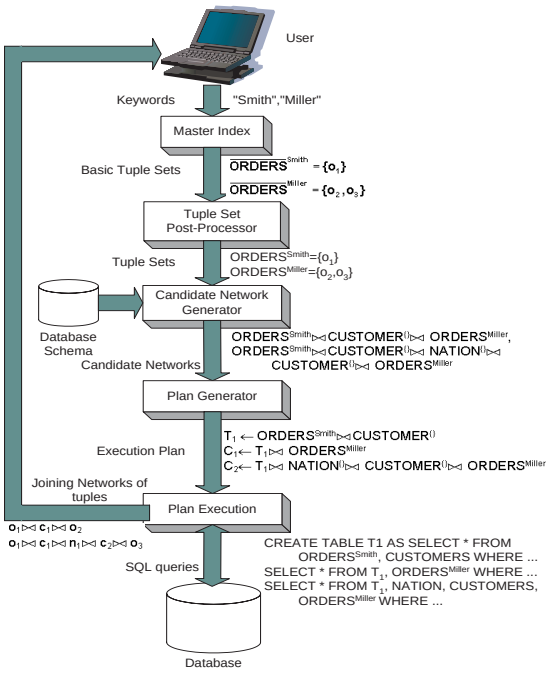


Figure 3: Architecture of DISCOVER

the minimal joining networks.

Then DISCOVER evaluates the candidate networks. Due to the nature of the problem, the candidate networks share join expressions. This offers an opportunity to build a set of intermediate results and use them in the computation of multiple candidate networks. The Plan Generator produces an execution plan that calculates and uses intermediate results in evaluating the candidate networks. Finally an SQL statement is produced for each line of the execution plan and these statements are passed to the DBMS. The DBMS returns the joining networks of tuples that are the solutions to the problem.

Notice that the candidate networks may have a number of joins that is only bound by the dataset as it is explained later. In these cases the user supplies a maximum number T of joins and DISCOVER incrementally outputs all candidate networks up to size T.

The challenges involved in the above process and the contributions of this paper are the following:

- We formalize keyword search on relational databases and provide intuitive semantics.
- We propose a modular architecture and have implemented DISCOVER based on it.
- We present an efficient candidate network generation algorithm. The naive approach would be to generate all join expressions up to size T that contain all keywords and then evaluate them. However, we prune out many of them by exploiting the properties of the schema of the database and the information returned by the master index. For example in the keyword query "Smith, Miller", the candidate network $ORDERS^{Smith} \bowtie CUSTOMER^{\{\}} \bowtie ORDERS^{Miller} \bowtie LINEITEM^{\{\}}$ is pruned out because $LINEITEM^{\{\}}$

has no keywords and since it is in the end of the joining sequence's chain, it cannot help in joining any tuple that could lead to a keyword. For more complex reasons, pertaining to the structure of the primary key to foreign key relationships as discussed later, candidate networks such as $ORDERS^{Smith} \bowtie LINEITEM^{\{\}} \bowtie ORDERS^{Miller}$ are also excluded.

- We prove that the candidate network generation algorithm creates a complete and non-redundant set of candidate networks, where "complete" means that the set of candidate networks produces all minimal joining networks of tuples (up to a given size T) and "non-redundant" means that if any candidate network of the set is excluded then there are database instances where there are minimal joining networks of tuples that are not discovered. It is also shown that the results of the candidate networks are always minimal joining networks of tuples.
- We specify when the maximum size of the candidate networks is bound by the size of the database schema and when it is bound only by the size of the database instance. For the former case, we provide theorems that specify the maximum size T_{max} of the minimal joining networks of tuples, as a function of the database schema.
- We propose a cost model. The Plan Generator module uses intermediate results to minimize the total cost of the evaluation of all candidate networks. We show that the problem of selecting the optimal set of intermediate results is NP-complete on the size of the candidate networks. We then present a tunable greedy algorithm that discovers near-optimal plans, without suffering from the unacceptable optimization time cost incurred by the optimal planning algorithm.
- DISCOVER has been implemented on top of Oracle 8i. We present a detailed experimental evaluation of the modules of DISCOVER and of the overall system. It is shown that a large percentage of the generated candidate networks are pruned. Furthermore, we show how to tune the greedy algorithm to achieve the best possible performance and we show that the overall performance beats by far the performance of the obvious straightforward approaches.

In Section 2 we compare DISCOVER to other related efforts. Sections 4 and 5 present the Candidate Network Generator and the Plan Generator module respectively. In Section 6 we evaluate experimentally the performance of DISCOVER. Finally, in Section 7 we conclude and discuss future extensions and improvements of DISCOVER.

2 Related Work

A framework for keyword search on databases when the schema is not known to the user is presented in [MV00b, MV00a]. An extension of SQL called Reflective SQL (RSQL) is introduced, which treats data and queries uniformly. The main limitation of this work is that all keywords must be contained in the same tuple. That is, the relationships between tuples from different relations are not taken into consideration.

In [GSVGM98] and [BNH+02], a database is viewed as a graph with objects/tuples as nodes and relationships as edges. Relationships are defined based on the properties of each application. For example an edge may denote a primary to foreign key relationship. In [GSVGM98], the user query specifies two sets of objects, the *Find* and the *Near* objects. These objects may be generated from two corresponding sets of keywords. The system ranks the objects in *Find* according to their distance from the objects in *Near*. An algorithm is presented that efficiently calculates these distances by building hub indices. In [BNH+02], answers to keyword queries are provided by searching for Steiner trees [Ple81] that contain all keywords. Heuristics are used to approximate the Steiner tree problem. A drawback of these approaches is that a graph of the tuples must be created and maintained for the database. Furthermore, the important structural information provided by the database schema is ignored and their algorithms work on huge data graphs. In contrast, DISCOVER is tuned to keyword search on relational databases and uses the properties of the schema of the database. Its key algorithms work on the schema graph, which is much smaller than the data graph, and does not need to keep any extra data representations. It exploits the properties of the database schema to produce the minimum number of SQL queries needed to answer to the keyword query. Furthermore, DISCOVER operates directly on the databases, so it does not have a main memory space limitation.

The work of the Candidate Network Generator reminds of algorithms for answering queries on universal relations [Ull82]. However there are many important differences between universal relations and DISCOVER: First, there is the obvious difference that the user of a Universal Relation (UR) needs to know the attributes where the keywords are, in contrast to the user of DISCOVER. Second, DISCOVER creates efficient queries that find all connections between the tuples that contain the keywords. In doing so, DISCOVER, unlike the UR, has to find connections whose size may not be schema bound and many of them are pruned by DISCOVER's Candidate Network Generator. Finally, in addition to finding the useful connections, DISCOVER exploits the fact that the connections are "correlated", in the sense that they share join expressions. This leads to a special query optimization algorithm, which is tuned to the specifics of our problem.

DBXplorer [ACD02] describes a multi-step system to answer keyword queries in relational databases and frees the user from the first limitation of the universal relations. However it does not consider solutions that include two tuples from the same relation. Furthermore they only consider exact matches, where a keyword must match exactly an attribute value and they do not exploit the reusability opportunities of the join trees, which is a simplified notion close to the candidate networks of DISCOVER.

Oracle 9i Text ([Ora01]) and IBM DB2 Text Informa-

tion Extender ([DB201]) use standard SQL to create full text indices on text attributes of relations. Microsoft SQL Server 2000 ([MSD01]) also provides tools to generate full text indices, which are stored in files outside the database. In all three systems, the user creates full text indices on single attributes and then performs keyword queries, which return the tuples that contain a keyword. Furthermore, keyword proximity queries are supported within a single attribute of a tuple, but not across different attributes or tuples. As we discuss, generalizing keyword search to work across tuples is very challenging and the issues are different from the text indexing issues that those systems address.

One of the criteria that we use to decide that a join expression J is not a candidate network is whether the joining networks of tuples produced by J contain more than one occurrences of the same tuple. Our approach for deciding this property can be viewed as a special case of the chase technique with inclusion dependencies presented in [AHV95]. Our algorithm is simpler, faster and decidable, since it focuses on primary to foreign key relationships.

Keyword search has been well studied for document databases ([Sal89]). For example [BP98] presents the Google search engine. [ACGM+01] offers an overview of current Web search engine design. It also introduces a generic search engine architecture and covers crawling and indexing issues. In [TWW+00], algorithms , data structures, and software are presented that approach the speed of keyword-based document search engines for queries on structural databases like parse trees, molecular diagrams and XML documents. [FKM99] tackles the keyword search problem in XML databases. They propose an extension to XML query languages that enables keyword search at the granularity of XML elements, which helps novice users formulate queries, but do not consider keyword proximity search.

The use of common subexpressions by the Plan Generator is a form of multi-query optimization [Sel88, Fin82, RSSB00]. However the candidate networks in DISCOVER have special properties that allow us to develop a more straightforward and efficient algorithm. The first property is that the candidate networks have small relations [Ull82] as leaves, which dramatically prunes the space of useful common subexpressions when applying the Wong-Yusefi algorithm [Ull82]. Second, the candidate networks are not random queries, but share common subexpressions by the nature of their generation as we see in Section 4. The techniques of [Fin82] cannot be applied to DISCOVER since they concentrate on finding common subexpressions as a post-phase to query optimization and DISCOVER does not have access to the DBMS optimizer.

3 Framework

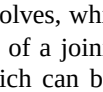
3.1 Data Model and Keyword Queries

We consider a database that has n relations $R_1, \ldots, R_n$. Each relation R_i has m_i attributes $a^i_1, \ldots, a^i_{m_i}$. The *schema graph* G is a directed graph that captures the primary key to foreign key relationships in the database schema. It has a node R_i for each relation R_i of the database and an edge $R_i \rightarrow R_j$ for each primary key to foreign key relationship from a set of attributes $(a^i_{b_1}, \ldots, a^i_{b_l})$ of R_i to a set of attributes $(a^j_{b_1}, \ldots, a^j_{b_l})$ of R_j, where $a^i_{b_k} \equiv a^j_{b_k}$ for $k = 1, \ldots, l$. We define the graph G_u to be the undirected version of G.

For notational simplicity, we assume that the attributes of a primary to foreign key relationship have the same name and that there are no self loops or parallel edges in the schema graph. So an edge $R_i \rightarrow R_j$ uniquely identifies the corresponding primary and foreign key attributes. We also assume that no set of attributes of any relation is both a primary key and a foreign key for two other relations, which is a reasonable assumption for any realistic database schema design. The generalization of the problem and the solution when these assumptions do not hold is trivial.

We denote the primary key of a tuple $t \in R$ as $p(t)$ and its foreign key that references relation S as $f_S(t)$.

Definition 1 (Joining network of tuples) *A joining network of tuples j is a tree of tuples where for each pair of adjacent tuples $t_i, t_j \in j$, where $t_i \in R_i, t_j \in R_j$, there is an edge (R_i, R_j) in G_u and $(t^i \bowtie t^j) \in (R_i \bowtie R_j)$.*

The *size* of a joining network is the number of joins that it involves, which is one less than the tree's size. An example of a joining network of tuples in Figure 2 is 

, which can be written in line notation as $c_1[o_1, o_2, n_1]$ or $o_1[c_1[o_2, n_1]]$. Its size is 3.

A *joining sequence of tuples* is a special case of a joining network of tuples, where each internal node of the tree has exactly two adjacent nodes. An example of a joining sequence of tuples in Figure 2 is $c_1[o_1, o_2]$, also denoted as $o_1 \bowtie c_1 \bowtie o_2$.

Definition 2 (Keyword Query) *A keyword query is a set of keywords $k_1, \ldots, k_m$. The result of the keyword query is the set of all possible joining networks of tuples that are both:*

- *Total: every keyword is contained in at least one tuple of the joining network.*
- *Minimal: we can not remove any tuple from the joining network and still have a total joining network of tuples.*

We call such joining networks Minimal Total Joining Networks of Tuples (MTJNT) *of the keywords $k_1, \ldots, k_m$ or simply MTJNT's when the corresponding set of keywords is obvious from the context.*

It is obvious from the definition that the result of a keyword query is unique. A keyword query may also be given the maximum size T of the result $MTJNT$'s.

The answers to keyword queries with two keywords are always joining sequences of tuples. On the other hand, when we have more than two keywords then the answer most often cannot be expressed as a joining sequence, so we need a joining network. Consider for example the keyword query "Smith, Miller, USA". The best (smallest) answer to this query is $MTJNT$ $c_1[o_1, o_2, n_1]$.

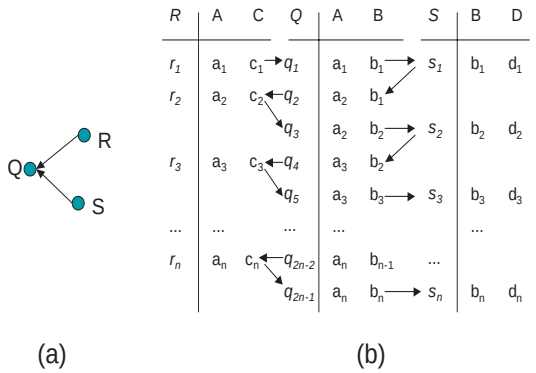


Figure 4: A many-to-many relationship

If there is a many-to-many relationship between two relations of a database, then the $MTJNT$'s could have an arbitrarily big size, which is only data bound. Figure 4 shows an extreme case where there is a many-to-many relationship between two relations R,S. There is a foreign to primary key relationship from Q to R and from Q to S on the homonymous attributes. Suppose that attribute values c_1 and d_n contain the two keywords of a query. The $MTJNT$ $r_1 \bowtie q_1 \bowtie s_1 \bowtie q_2 \bowtie r_2 \bowtie \ldots \bowtie r_k \bowtie q_{2k-1} \bowtie s_k \bowtie q_{2k} \bowtie \ldots \bowtie r_n \bowtie q_{2n-1} \bowtie s_n$ uses all tuples from all three relations, as shown by the arrows in Figure 4. So we see that the size of the joining sequence can only be bound by the dataset when there are many-to-many relationships.

Keep in mind that a keyword may be in more than one tuples of the same relation or in different relations. An example of the first case is the "Miller" keyword that appears in two tuples (o_2 and o_3) of the *ORDERS* relation. For the second case, consider the keyword query "John, USA", where the keyword "John" is contained in both tuples o_1 and c_3. Two joining sequences for this keyword query are $c_3 \bowtie n_1$ and $o_1 \bowtie c_1 \bowtie n_1$. Notice that the joining sequences in the result in this case are heterogeneous.

3.2 Architecture

In this section we walk through the components of DISCOVER (see Figure 3) and formally define the structure of their inputs and outputs. The *Master Index* inputs a set of keywords $k_1,\ldots,k_m$ and outputs a set of *basic tuple sets* $\overrightarrow{R_i^{k_j}}$ for $i = 1,\ldots,n$ and $j = 1,\ldots,m$. The basic tuple set $\overrightarrow{R_i^{k_j}}$ consists of all tuples of relation R_i that contain the keyword k_j. The master index has been implemented using the Oracle8i interMedia Text 8.1.5 extension, which builds full text indices on single attributes of relations. Then the master index inspects the index of each attribute and combines the results. [1]

Then the *Tuple Set Post-Processor* takes the basic tuple sets and produces tuple sets R_i^K for all subsets K of $\{k_1,\ldots,k_m\}$, where

$$R_i^K = \{t | t \in R_i \wedge \forall k \in K, t \text{ contains } k \wedge \forall k \in \{k_1,\ldots,k_m\} - K, t \text{ does not contain } k\} \quad (1)$$

i.e., R_i^K contains the tuples of R_i that contain all keywords of K and no other keywords.

The tuple sets are obtained from the basic tuple sets using the following formula.

$$R_i^K = \bigcap_{k \in K} \overrightarrow{R_i^k} - \bigcup_{k \in \{k_1,\ldots,k_m\} - K} \overrightarrow{R_i^k} \quad (2)$$

The non-empty tuple sets along with the schema graph of the database are passed to the *Candidate Network Generator*. For brevity reasons that become clear below, we will call the database relations, which appear in the schema graph, *free tuple sets*. They are denoted as $R^{\{\}}$.

Definition 3 (Joining Network of Tuple Sets) *A joining network of tuple sets J is a tree of tuple sets where for each pair of adjacent tuple sets R_i^K, R_j^M in J there is an an edge (R_i, R_j) in G_u.*

For example a joining network of tuple sets for the database of Figures 1, 2 is $CUSTOMER^{\{\}}$ $[ORDERS^{Smith}, ORDERS^{Miller}, NATION^{\{\}}]$. A joining sequence of tuple sets is a special case of joining networks of tuples, where each intermediate node of the tree has exactly two adjacent nodes and it is denoted as $TS_1 \bowtie \ldots \bowtie TS_l$. We say that a joining network of tuples j belongs to a joining network of tuple sets J ($j \in J$) if there is a tree isomorphism mapping h from the tuples of j to the tuple sets of J, such that for each tuple $t \in j$, $t \in h(t)$. For example, in the instance of 2, $c_1[o_1, o_2, n_1] \in CUSTOMER^{\{\}}$ $[ORDERS^{Smith}, ORDERS^{Miller}, NATION^{\{\}}]$.

DISCOVER does not generate any joining networks of tuple sets that are redundant or cannot produce any MTJNT's. We call the joining networks of tuple sets generated by DISCOVER *candidate networks*.

Definition 4 (Candidate Network) *Given a set of keywords $k_1,\ldots,k_m$, a candidate network C is a joining network of tuple sets, such that there is an instance I of the database that has a $MTJNT$ $M \in C$ and no tuple $t \in M$ that maps to a free tuple set $F \in C$ contains any keywords.*

We need the last condition to make sure that no keywords are accidentally added to M. Such a MTJNT will also belong to a candidate network that has a non-empty tuple set instead of F, so C is redundant. For example, consider the database instance of Figure 2 and the keyword query "Smith, Miller". $J = ORDERS^{Smith} \bowtie CUSTOMER^{\{\}} \bowtie ORDERS^{\{\}}$ is not a candidate network even though the MTJNT $o_1 \bowtie c_1 \bowtie o_2$ belongs to J. J is subsumed by $ORDERS^{Smith} \bowtie CUSTOMER^{\{\}} \bowtie ORDERS^{Miller}$.

There are many joining networks of tuple sets that are not candidate networks. For example, the network $J = ORDERS^{Smith} \bowtie LINEITEM^{\{\}} \bowtie ORDERS^{Miller}$ is not a

[1] We are currently building from scratch a more efficient master index using an inverted index that has one entry for each keyword k and the entry has references to all tuples that contain k. However, the master index choice does not affect the key challenges and tradeoffs discussed in this paper.

candidate network because there is no joining network of tuples $j = o^S \bowtie l \bowtie o^M$ where $j \in J$ and $o^S \not\equiv o^M$. We will analyze the conditions that promote a network into a candidate network in Section 4.

Each candidate network of size N will produce zero or more $MTJNT$'s of size N. The result of the keyword search is the union of the $MTJNT$'s produced by all possible candidate networks.

The set of candidate networks is passed to the *Plan Generator*, which optimizes the evaluation of the candidate networks.

Definition 5 (Execution Plan) *Given a set $C_1, \ldots, C_r$ of candidate networks, an execution plan is a list $A_1, \ldots, A_s$ of assignments of the form $H_i \leftarrow B_{i_1} \bowtie \ldots \bowtie B_{i_t}$ where:*

- *Each B_{i_j} is either a tuple set or an intermediate result defined in a previous assignment. The latter requires that there is an index $k < i$, such that $H_k \equiv B_{i_j}$.*
- *For each candidate network C there is an assignment A_i, that computes C.*

For example, an execution plan for the keyword query shown in Figure 3 is

$T_1 \leftarrow ORDERS^{Smith} \bowtie CUSTOMER^{\{\}}$,
$C_1 \leftarrow T_1 \bowtie ORDERS^{Miller}$,
$C_2 \leftarrow T_1 \bowtie NATION^{\{\}} \bowtie CUSTOMER^{\{\}} \bowtie ORDERS^{Miller}$

where T_1 is an intermediate result. The number of joins of this plan is 5, whereas the number of joins to evaluate the two candidate networks without building any intermediate results would be 6. As the number of candidate networks increases the difference in the number of joins increases dramatically.

Finally the execution plan is passed to the *Plan Execution* module, which translates the assigments of the plan to SQL statements. The assignments that build intermediate results are translated to "CREATE TABLE" statements and the candidate network evaluation assigmnents to "SELECT-FROM-WHERE" statements. The union of the results of these "SELECT-FROM-WHERE" statements is the result of the keyword search and it is returned to the user. The smaller MTJNT's are returned first.

4 Candidate Network Generation

The Candidate Network Generator inputs the set of keywords $k_1, \ldots, k_m$, the non-empty tuple sets R_i^K and the maximum candidate networks' size T and outputs a complete and non-redundant set of candidate networks. The key challenge is to avoid the generation of redundant joining networks of tuple sets. The solution to this problem requires an analysis of the conditions that force a joining network of tuples to be non-minimal - the condition for the totality of the network is straightforward. Then the candidate networks generation algorithm is presented and we present theorems that show that it is (i) complete, ie., every $MTJNT$ is produced by a candidate network output by the algorithm, and (ii) it does not produce any redundant candidate networks. Finally we give an example of the algorithm's execution steps.

We must ensure that the joining networks of tuples that belong to a candidate network are total and minimal. The condition that a joining network of tuple sets J must satisfy in order to ensure totality of the produced joining networks of tuples $j \in J$ is to contain all keywords. That is,

$$\forall k \in \{k_1, \ldots, k_m\}, \exists R_i^K \in J, k \in K \tag{3}$$

For example $ORDERS^{Smith} \bowtie CUSTOMER^{\{\}} \bowtie ORDERS^{\{\}}$ is not total with respect to the keyword query "Smith, Miller". Equation 3 does not ensure minimality. There are two cases when a joining network of tuples j is not minimal.

1. A joining network of tuples j is not minimal if it has a tuple with no keywords as a leaf. In this case we can simply remove this leaf. We carry this condition to joining networks of tuple sets by not allowing free tuple sets as leaves. For example $ORDERS^{Smith} \bowtie CUSTOMER^{\{\}} \bowtie ORDERS^{Miller} \bowtie CUSTOMER^{\{\}}$ is rejected since it has the free tuple set $CUSTOMER^{\{\}}$ as a leaf.

2. j is not minimal if it contains the same tuple t twice. In this case we can collapse the two occurrences of t. We carry this condition to joining networks of tuple sets by detecting networks that are bound to produce non-minimal joining networks of tuples, regardless of the database instance. According to this condition, the joining network of tuple sets $J = ORDERS^{Smith} \bowtie LINEITEM^{\{\}} \bowtie ORDERS^{Miller}$ is ruled out because the structure of J ensures that all the produced joining networks of tuples $j = o^S \bowtie l \bowtie o^M$ will contain the same tuple twice. To see this suppose that o^S has primary key $p(o^S)$. It is joined with l, so l has foreign key $f_{ORDERS}(l) = p(o^S)$. l will also join with $o^M \in ORDERS^{Miller}$. So, it is $p(o^M) = f_{ORDERS}(l) = p(o^S)$. Hence $o^M \equiv o^S \equiv o^{M,S}$ and j cannot be minimal.

Theorem 1 presents a criterion that determines when the joining networks of tuples produced by a joining network of tuple sets J have more than one occurrences of a tuple.

Theorem 1 *A joining network of tuples j produced by a joining network of tuple sets J has more than one occurrences of the same tuple for every instance of the database if and only if J contains a subgraph of the form R^K—S^L—R^M, where R, S are relations and there is an edge $R \rightarrow S$ in the schema graph.*

Hence, we conclude to the following criterion.

Criterion 1 (Pruning Condition) *A candidate network does not contain a subtree of the form R^K—S^L—R^M, where R and S are relations and the schema graph has an edge $R \rightarrow S$.*

```
Algorithm Candidate Networks Generator
Input: tuple set graph G_TS, T, k_1,...,k_m
Output: set of candidate networks with size up to T
{
  Q: queue of joining networks of tuple sets
  Pick a keyword k_t ∈ {k_1,...,k_m}
  for each tuple set R_i^K where i = 1,...,n and k_t ∈ K do
    Add joining networks of tuple sets R_i^K to Q
  while Q not empty do {
    Get head C from Q
    if C satisfies the pruning condition then ignore C
    else if C satisfies the acceptance conditions then output C
    /*There is no reason to extend accepted joining networks of tuple sets*/
    else
      for each tuple set R_i^K adjacent in G_TS (ignoring edge direction) to a node of C
        if (K = {} OR ∄R_j^M ∈ (C ∪ R_i^K), M ≠ {} ∧ keywords(C ∪ R_i^K) = keywords((C ∪ R_i^K) − R_j^M))
    /*Expansion rule*/
          and (size of C < T) then {
          if R_i^K is adjacent to R_j^M in C = R_j^M[...] then C ← R_i^K[R_j^M[...]]
          Put C in Q
        }
        else ignore R_i^K
  } }
```

Figure 5: Algorithm for generating the candidate networks

4.1 Candidate Networks Generation Algorithm

The candidate network generation algorithm is shown in Figure 5. First, we create the *tuple set graph* G_{TS}. A node R_i^K is created for each non-empty tuple set R_i^K, including the free tuple sets. An edge $R_i^K \rightarrow R_j^M$ is added if the schema graph G has an edge $R_i \rightarrow R_j$. The algorithm is based on a breadth-first traversal of G_{TS}. We keep a queue Q of "active" joining networks of tuple sets. In each round we pick from Q an active joining network of tuple sets J and either (i) discard J because of the pruning condition (Criterion 1) or (ii) output J as a candidate network or (iii) expand J into larger joining networks of tuple sets (and place them in Q). We start the traversal from all tuple sets that contain a randomly selected keyword $k_t \in \{k_1, \ldots, k_m\}$.

An active joining network of tuple sets C is expanded according to the following *expansion rule*: A new active joining network of tuple sets is generated for each tuple set R_i^K, adjacent to C in G_{TS}, if either R_i^K is a free tuple set ($K = \{\}$) or after the addition of R_i^K to C every non-free tuple set of C (including R_i^K) contributes at least one keyword that no other non-free tuple set contributes, i.e.,

$$\nexists R_j^M \in (C \cup R_i^K), M \neq \{\} \wedge \\ keywords(C \cup R_i^K) = keywords((C \cup R_i^K) - R_j^M)$$

where $keywords(J)$ returns the union of keywords in the tuple sets of the joining network of tuple sets J, i.e., $keywords(J) = \bigcup_{R_i^K \in J} K$. A free tuple set may be visited more than once. Each non-free tuple set is used at most once in each candidate network. The reason is that, for all database instances, the result of a joining network of tuple sets J with two occurrences of the same non-free tuple set R_i^K is subsumed by the result of a joining network of tuple sets J', generated by the algorithm, that is identical to J but has $R_i^{\{\}}$ instead of the second occurrence of R_i^K.

In addition, the implementation never places in Q a joining network of tuples sets J that has more than m leaves, where m is the number of keywords in the query. For example, if the keywords are two then only joining sequences are placed in Q. Indeed, even if this rule were excluded the output of the algorithm would be the same, since such a network J can neither meet the acceptance conditions listed next nor be expanded into a network J' that meets the acceptance conditions. Nevertheless, the rule leads to cleaner traces and better running time.

The algorithm outputs a joining network of tuple sets J if it satisfies the following *acceptance conditions*:

- The tuple sets of J contain all keywords, i.e., $keywords(J) = \{k_1, \ldots, k_m\}$.
- J does not contain any free tuple sets as leaves.

An important property of the algorithm is that it outputs the candidate networks with increasing size. That is, the smaller candidate networks, which are the better solutions to the keyword search problem, are output first.

Theorems 2 and 3 prove the completeness and the minimality of the results of the algorithm.

Theorem 2 (Completeness) *Every solution of size T to the keyword query is produced by a candidate network of size T, output by the candidate network generator.*

Theorem 3 (No Redundancy) *For each candidate network C output by the algorithm, given the tuple set graph*

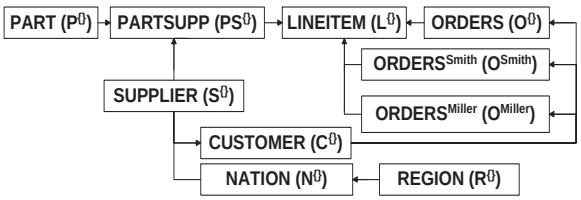


Figure 6: Tuple set graph

#	*Queue/from/candidate networks output*
1a	O^{Smith}
2a	$O^{Smith} \bowtie L^{\{\}}$/1a
b	$O^{Smith} \bowtie C^{\{\}}$/1a
3a	$O^{Smith} \bowtie L^{\{\}} \bowtie O^{\{\}}$(pruned)/2a
b	$O^{Smith} \bowtie L^{\{\}} \bowtie O^{Miller}$(pruned)/2a
c	$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}}$/2a
d	$O^{Smith} \bowtie C^{\{\}} \bowtie O^{\{\}}$/2b
e	$O^{Smith} \bowtie C^{\{\}} \bowtie O^{Miller}$/2b
f	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}}$/2b
4a	$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}} \bowtie P^{\{\}}$/3c/ $O^{Smith} \bowtie C^{\{\}} \bowtie O^{Miller}$
b	$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}} \bowtie L^{\{\}}$/3c
c	$O^{Smith} \bowtie C^{\{\}} \bowtie O^{\{\}} \bowtie C^{\{\}}$(pruned)/3d
d	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}}$/3f
	...
5a	$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}} \bowtie P^{\{\}} \bowtie PS^{\{\}}$(pruned)/4a
b	$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}} \bowtie L^{\{\}} \bowtie O^{Miller}$/4b
c	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}} \bowtie O^{Miller}$ /4d
d	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}} \bowtie O^{\{\}}$/4d
e	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}} \bowtie N^{\{\}}$(pruned)/4d
	...
6a	$O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}} \bowtie O^{\{\}} \bowtie C^{\{\}}$(pruned)/5d/ $O^{Smith} \bowtie C^{\{\}} \bowtie N^{\{\}} \bowtie C^{\{\}} \bowtie O^{Miller}$
	.../ /$O^{Smith} \bowtie L^{\{\}} \bowtie PS^{\{\}} \bowtie L^{\{\}} \bowtie O^{Miller}$
7	...

Figure 7: Example

G_{TS}[2], there is an instance I of the database that produces the same tuple set graph G_{TS}, contains a MTJNT $j \in C$ and j does not belong to any other candidate network.

Example. We present the execution of the candidate network generator algorithm for the keyword query "Smith, Miller" on the TPC-H schema and the database instance in Figure 2, for $T = 5$. That is, we consider candidate networks having at most 5 joins. The tuple set graph is shown in Figure 6.

Suppose we pick "Smith" as the k_t of the algorithm. Hence we put $ORDERS^{Smith}$ into the queue. The state of the queue and the candidate networks output in each iteration are shown in Figure 7. We use the obvious abbreviated names for the relations. Since the query has only two keywords, only joining sequences are generated and eventually output.

Maximum size of candidate networks. Depending on the form of the database schema, the maximum size T_{max} of the candidate networks may be *bounded* or *unbounded* by the database schema.

Theorem 4 *T_{max} is unbounded if and only if G has one of the following properties:*

- *There is a node of G that has at least two incoming edges.*
- *G has a directed cycle.*

5 Evaluation of Candidate Networks

The *Plan Generator* module of DISCOVER inputs a set of candidate networks and creates an execution plan to evaluate them as defined in Section 3.2. The key optimization opportunity is that typically the candidate networks share join subexpressions. Efficient execution plans store the common join expressions as intermediate results and reuse them in evaluating the candidate networks. For example, in Figure 3 we calculate and store the join expression $ORDERS^{Smith} \bowtie CUSTOMER^{\{\}}$. $CUSTOMER^{\{\}} \bowtie ORDERS^{Miller}$ is also a common join expression but it will not help to store both, as we explain below.

The space of execution plans that can be generated for a set of candidate networks is huge. We prune it by the following two assumptions: First, we define every non-free tuple set to be a *small* relation, since its tuples are restricted to contain specific keywords. The result of a join that involves a small relation is also a small relation. Those assumptions lead to the conclusion that every join expression of the plan must contain a small relation and, hence, all intermediate results are small. Note that both the assumptions and the conclusion follow directly the Wong-Yousefi algorithm ([Ull82]) of INGRES. Indeed, in practice, the intermediate results are sufficiently small to be stored in main memory as we discuss in Section 6.

Second, the plan generator only considers plans where the right hand side of the assignments $H_i \leftarrow B_{i_1} \bowtie \ldots \bowtie B_{i_t}$ of Definition 5 in Section 3.2 are joins of exactly two arguments, i.e., $t = 2$. This policy is based on the assumption that the cost of calculating and storing the results of both $A \bowtie B$ and $A \bowtie B \bowtie C$ is essentially the same with the cost for just calculating and storing the result of $A \bowtie B \bowtie C$, if the DBMS optimizer selects to first calculate $A \bowtie B$ and then the result of $A \bowtie B \bowtie C$. Hence we can store and possibly reuse later $A \bowtie B$ "for free".

This assumption is very precise when there are indices on the primary and foreign key attributes. Then the joins (and, in particular, the most expensive ones) are executed in a series of index-based 2-way joins. The assumption always held for the Oracle 8i DBMS that we used in our TPC-H-based experimentation. (The assumption deviates from reality when there are no indices and the database chooses multi-way merge-sort joins.)

In summary, the plan generator considers and evaluates the space of plans where the joins have exactly two arguments. Note that once a plan P is selected from the restricted space we outlined, the plan generator eliminates

[2]Notice that the candidate network generator does not examine the tuples of a specific tuple set, but only whether it is empty or not.

non-reused intermediate results by inlining their definition into the single point where they are used. That is, given two assignments

$$T \leftarrow A \bowtie B$$
$$T' \leftarrow T \bowtie C$$

if T is not used at any other place than the computation of T', the two assignments will be merged into

$$T' \leftarrow A \bowtie B \bowtie C$$

Cost Model. The theoretical study of the complexity of selecting the optimal execution plan is based on a simple cost model of execution plans: We assign a cost of 1 to each join. We use this theoretical cost model in proving that the selection of the optimal execution plan is NP-complete (Theorem 5). It is easy to see that the problem is also NP-hard for the actual cost model of DISCOVER.

The actual cost model of DISCOVER exploits the fact that we can get the sizes of the non-free tuple sets from the master index. We also assume that we know the selectivity of the primary to foreign key joins, which can be calculated from the sizes of the relations. The actual cost model defines the cost of a join to be the size of its result in number of tuples. (The cost model can easily be modified to work for the size in bytes instead of the number of tuples.) The cost of the execution plan is the sum of the costs of its joins. Notice that since there are indices on the primary and foreign keys, the cost of a join is proportional to the size of its result, since the joins will typically be index-based joins.

The problem of deciding which intermediate results to build and store can be formalized as follows:

Problem 1 (Choice of intermediate results) *Given a set of candidate networks, find the intermediate results that should be built, so that the overall cost of building these results and evaluating the candidate networks is minimum.*

Theorem 5 shows that Problem 1 is NP-complete on the size of the candidate networks with respect to the theoretical cost model defined above.

Theorem 5 *Problem 1 is NP-complete.*

5.1 Greedy algorithm

Figure 8 shows a greedy algorithm that produces a near-optimal execution plan, with respect to the actual cost model defined above, for a set of candidate networks by choosing in each step the join m between two tuple sets or intermediate results that maximizes the quantity $\frac{frequency^a}{log^b(size)}$, where $frequency$ is the number of occurences of m in the candidate networks, $size$ is the estimated number of tuples of m and a, b are constants. The $frequency^a$ term of the quantity maximizes the reusability of the intermediate results, while the $log^b(size)$ term minimizes the size of the intermediate results that are computed first. We have experimented with multiple combinations of values for a and b and found that the optimal solution is closer approximated

```
Algorithm Select list of intermediate results
Input: set S of candidate networks of size ≤ T
Output: list L of intermediate results to build
{
  while not all candidate networks in S have
    been added to L do {
    Let Z be the set of all small join
      subexpressions of 1 join contained in
      at least one candidate network in S;
    Add the intermediate result m with the
      maximimum frequency^a / log^b(size) value in Z to L;
    Rewrite all candidate networks in S to use
      m where possible;
  }
}
```

Figure 8: Greedy algorithm for selecting a list of intermediate results to build

for $\{a, b\} = \{1, 0\}$, when the size of the candidate networks (and the reusability) increases.

We perform a worst case time analysis of the greedy algorithm. The while loop is executed at most $|S| \cdot T$ times if every join has a frequency of 1, where $|S|$ is the number of candidate networks. The calculation of Z takes time $|S| \cdot T$. We assume that we traverse a candidate network of size T_1 in time $O(T_1)$. In each step, we keep a hash table H with each intermediate result in Z and its frequency. Hence we check if an intermediate result is already in H and increase its frequency in $O(1)$. Finding the intermediate result in H that maximizes $\frac{frequency^a}{log^b(size)}$ takes time $|S| \cdot T$. The rewriting step also takes time $|S| \cdot T$. Hence the total execution time takes in the worst case time $O((|S| \cdot T)^2)$.

The greedy algorithm may output a non optimal list of intermediate results. However, in special cases the greedy is guaranteed to produce the optimal plan. One such case is described by the theorem below:

Theorem 6 *The greedy algorithm for $(a, b) = (1, 0)$ is optimal for $m = 2$ keywords, when each of them is contained in exactly one relation.*

6 Experiments

We evaluate the algorithms of DISCOVER with detailed performance evaluation on a TPC-H database. First we measure the pruning efficiency of the candidate network generator. In particular, we measured how many joining networks of tuple sets are ruled out based on the pruning conditions of the candidate network generator. Then we compare the plans produced by the greedy to the ones produced by the optimal, where the optimal execution plan is computed using an exhaustive algorithm. We also compare the speedup in runtime performance for generating and executing the execution plan using the greedy and the optimal algorithm compared to the naive method, where no intermediate results are built. Finally, we compare the overall execution times of DISCOVER for some typical

#keyw	JNTS_K	JNTS_L	CNs	neTS's
2	25	5.355	4.485	2.96
3	55.22	13.86	9.27	4.35
4	85.69	33.88	24.03	5.91
5	101	37.3	26	7.12

(a) Fix maximum candidate networks' size to 3

MaxCNsize	JNTS_K	JNTS_L	CNs	neTS's
1	0.95	0.95	0.95	2.96
2	3.72	2.36	2.12	2.96
3	29.22	4.74	3.7	2.96
4	422.88	10.36	6.4	2.96
5	6941	24.75	11.45	2.96

(b) Fix number of keywords to 2

MaxCNsize	JNTS_K	JNTS_L	CNs	neTS's
1	0.59	0.59	0.59	4.35
2	5.01	3.91	3.35	4.35
3	55.22	13.86	9.27	4.35
4	639.61	50.49	29.51	4.35
5	7532	223	103.66	4.35

(c) Fix number of keywords to 3

Figure 9: Evaluation of the candidate network generator

keyword queries to the naive method and to the optimal method.

We use the TPC-H database to conduct the experiments. The size of the database is 100MB. We use Oracle 9i, running on a Xeon 2.2GHz PC with 1GB of RAM. DISCOVER has been implemented in Java and connects to the DBMS through JDBC. The master index is implemented using the full-text Oracle9i interMedia Text extension. The basic tuple set of relation R for keyword k is produced by merging the tuples returned by the full-text index on each attribute of R. We found out that each keyword is contained on the average in 3.5 relations, that is, 3.5 non-empty basic tuple sets are created for each keyword.

The tuple sets and the intermediate results are stored in tables in the KEEP buffer pool of Oracle 9i, which retains objects in memory, thus avoiding I/O operations. We dedicated 70MB to the KEEP buffer pool. The display time is not included in the measured execution time.

The naive method does not produce any intermediate results – it simply executes each candidate network. The execution times for both the naive method and DISCOVER's evaluation method, which builds and reuses intermediate results, depend on the status of the cache of the DBMS. In order to eliminate this factor we warm-up the cache before executing the experiments. The warm-up is done by executing the SQL queries corresponding to the candidate networks produced by the candidate network generator. Hence, we are certain that the warm-up does not favor DISCOVER more than the naive method.

Evaluation of the candidate network generator. This experiment measures the pruning capabilities of the candidate network generator. We use the TPC-H schema but we do not use the TPC-H dataset in this experiment, because we want to control the distribution of the number of occurrences of each keyword. So, we randomly put the keywords of the keyword query in the relations. Each keyword is contained in a relation R with probability $a \cdot log(size(R))$, where $size(R)$ is the number of tuples in R as defined in the TPC-H specifications for scale factor SF=1. We selected $a = \frac{1}{10 \cdot log_2(6{,}000{,}000)}$. This means that the probability that a keyword is contained in the $LINEITEM$ relation, which is the largest one, is $\frac{1}{10}$, since $size(LINEITEM) = 6{,}000{,}000$. This probability is about $\frac{1}{100}$ for the $REGION$ relation, which is the smallest one. We measure three numbers of joining networks of tuple sets for each execution of the experiment.

1. $JNTS\_K$ is the number of joining networks of tuple sets of size up to T that have the following properties:
   - They contain all keywords of the keyword query, i.e., they are total.
   - No non-free tuple set can be replaced by a free tuple set and still have all keywords in the joining network of tuple sets.
2. $JNTS\_L$ is the number of joining networks of tuple sets that have only non-free tuple sets as leaves in addition to the above properties.
3. CNs is the number of candidate networks generated by DISCOVER. Those candidate networks have one more property in addition to the above properties; they do not produce joining networks of tuples with more than one occurrences of the same tuple (Criterion 1).

We also measure the number of non-empty basic tuple sets (*neTS's*) generated in each execution. Figure 9 shows the average results of the experiment for 1000 executions. Notice that the ratio $\frac{CNs}{JNTS\_L}$ decreases as the maximum size of the output candidate networks increases, i.e., Criterion 1 prunes more when the candidate networks are larger. The reason is that the trigger of Criterion 1 has more places to happen in a large candidate network.

#keyw	$\frac{Cost(O)}{Cost(G)}$
2	1
3	0.96
4	0.96
5	0.97

(a) Fix CN size to 3

MaxCN	$\frac{Cost(O)}{Cost(G)}$
1	1
2	1
3	0.96
4	0.93
5	0.90

(b) Fix # keywords to 3

Figure 10: Evaluation of the Plans of the Greedy Algorithm

Quality of Greedy. The quality of the plans produced by the greedy algorithm are very close to the quality of the plans produced by the optimal. We use the same settings with the above experiment. In Figure 10 we show how well the plans produced by the greedy algorithm perform on the average, compared to the optimal plans for $(a = 1, b = 0)$ for 50 executions. In about 70% of the cases the generated plans turn out to be identical and in the cases where they are different, the differences are fairly small.

Evaluation of Plan Generator. In this experiment we measure the speedup that DISCOVER's plan generator induces. In particular, we compare the time spent in DISCOVER's Plan Generator and Plan Execution modules against the baseline provided by the naive method. We

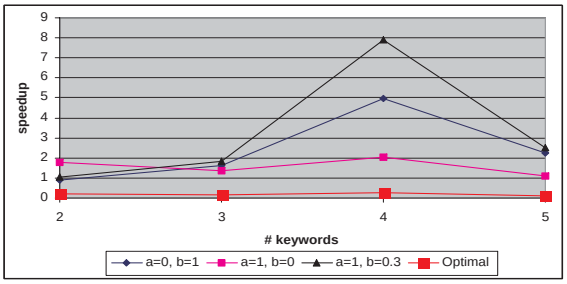
also compare the optimal method against the baseline of the naive method. In detail, the measured methods are:

1. *DISCOVER's method.* We calculate the execution plan using the greedy algorithm for three different combinations of values for a, b. In particular, $\{a,b\} \in \{(1,0),(0,1),(1,0.3)\}$. Recall that a and b are the weights we assign to the reusability and the size of the intermediate results respectively.
2. *Naive method.* We evaluate the candidate networks without using any intermediate results.
3. *Optimal method.* We calculate the optimal execution plan using an exhaustive algorithm.

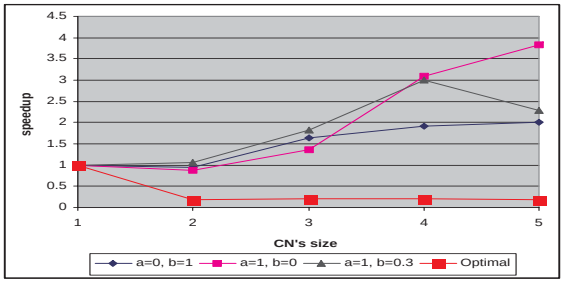
In each execution of the experiment we randomly select m keywords from the set of words that are contained in the TPC-H database. Then DISCOVER uses the master index to generate the tuple sets and the candidate network generator calculates the candidate networks of size up to T. The execution continues separately for each of the execution methods. We executed the experiment 200 times and measured the average speedup $\frac{Time(Naive)}{Time(other\ method)}$, which indicates that DISCOVER's methods (or the optimal) are $\frac{Time(Naive)}{Time(other\ method)}$ times faster than the naive.

The results are shown in Figure 11. The optimal method is always worse than the naive due to the great time overhead in discovering the optimal execution plan. Notice in Figure 11 (a) that the speedup decreases when the number of keywords is greater than 4, because there are more distinct tuple sets in the candidate networks and hence the reusability opportunities decrease since the candidate networks' size is fixed to 3. Also notice in Figure 11 (b) how the greedy algorithm with $\{a,b\} = \{0,1\}$ performs better than the one with $\{a,b\} = \{1,0\}$ when the sizes of the candidate networks are smaller than 4. This happens because the reusability opportunities increase as the size of the candidate networks increases, so the *frequency* factor of the greedy algorithm becomes dominant. In general, the $(a = 1, b = 0)$ and $(a = 1, b = 0.3)$ options perform better as the difference between the size of the candidate networks and the number of keywords increases, since this creates more opportunities for reusability.

Execution times. Finally, we measure the average absolute execution times to answer a keyword query using the three methods described above. The execution times in this experiment include the time to generate the candidate networks using DISCOVER's candidate network generator, but not the time to build the tuple sets, which takes from 2 to 4 seconds and could be considerably reduced by using a more efficient master index implementation [ACD02]. The TPC-H dataset is not suitable for this experiment, because it has less than 500 distinct keywords, which are repeated thousands of times. Hence, we inserted into the 100MB TPC-H database, 100 new tuples to each relation. These tuples contain 50 new keywords and each keyword is contained in exactly 50 tuples in two different relations (two non-empty basic tuple sets are created for each keyword). In each execution, the keyword query consists of two randomly selected keywords from the 50 new keywords. Figure 12 shows the average execution times for the three methods for 100 executions. Again, notice the superiority of the $(a = 1, b = 0)$ and $(a = 1, b = 0.3)$ methods when the size of the candidate networks increases, which happen also to be the toughest cases from a performance point of view. The $a = 1$ parameter leads the greedy to exploit the opportunities for reusing intermediate results.



(a) Fix maximum candidate networks' size to 3



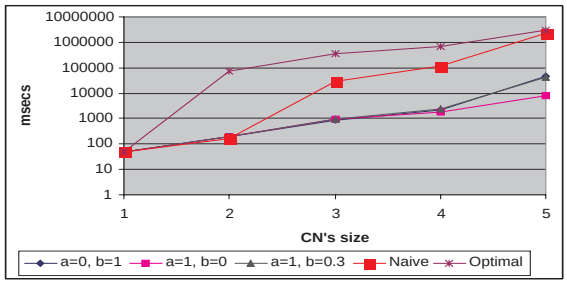
(b) Fix number of keywords to 3

Figure 11: Speedup when using intermediate results

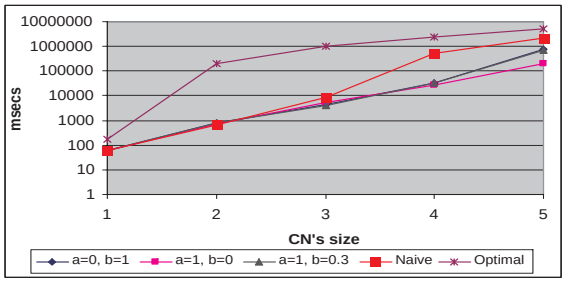
7 Conclusion and Future Work

As the amount of information stored in databases increases, so does the need for efficient information discovery. Keyword search enables information discovery without requiring from the user to know the schema of the database, SQL or some QBE-like interface, and the roles of the various entities and terms used in the query. in databases that do not require knowledge of the database schema or of a querying language. We presented DISCOVER, a system that performs keyword search in relational databases. It proceeds in three step. First it generates the smallest set of candidate networks that guarantee that all $MTJNT$'s will be produced. Then the greedy algorithm creates a near-optimal execution plan to evaluate the set of candidate networks. Finally, the execution plan is executed by the DBMS.

In this work, we defined two keywords to be associated if they are contained in two tuples connected through primary to foreign key relationships. This is an intuitive and challenging association criterion as we have shown. In the future, we plan to extend DISCOVER to handle more association criteria such as: First, the keywords may be part of the metadata of the database. For example the keyword query "customer, Lou" would return tuple c_1 in Figure 2. Second, we could define two keywords, which are contained in the same attribute of two tuples of a relation, to



(a) Fix number of keywords to 2



(b) Fix number of keywords to 3

Figure 12: Execution times

be associated. For example, the tuples o_1, o_2 are a solution to the keyword query "Smith, Miller", by this criterion.

We plan to apply new optimization techniques to DISCOVER. For example we plan to apply dynamic optimization techniques in the evaluation of the candidate networks. Currently, DISCOVER uses static optimization methods. That is, the whole execution plan is generated before its evaluation begins. Furthermore, we plan to experiment with new cost models that access the DBMS's optimizer. We are also building from scratch a more efficient master index.

Finally, we are working on building polynomial time algorithms that generate an optimal execution plan for special cases of database schemas.

8 Acknowledgements

We wish to thank Charles Elkan for providing the problem. We also thank Nick Koudas for the discussions we made and the ideas he proposed.

References

[ACD02] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: A System For Keyword-Based Search Over Relational Databases. *ICDE*, 2002.

[ACGM+01] Arvind Arasu, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. Searching the web. *Transactions on Internet Technology*, 2001.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. *Addison Wesley*, 1995.

[BNH+02] G. Bhalotia, C. Nakhey, A. Hulgeri, S. Chakrabarti, and S. Sudarshanz. Keyword Searching and Browsing in Databases using BANKS. *Proceedings of International Conference on Data Engineering*, 2002.

[BP98] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *WWW Conference*, 1998.

[DB201] http://www.ibm.com/software/data/db2/extenders/textinformation/index.html. 2001.

[Fin82] Sheldon J. Finkelstein. Common subexpression analysis in database applications. *ACM SIGMOD*, 1982.

[FKM99] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating Keyword Search into XML Query Processing. *WWW9 Conference*, 1999.

[GSVGM98] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. *VLDB*, 1998.

[MSD01] http://msdn.microsoft.com/library/. 2001.

[MV00a] U. Masermann and G. Vossen. Schema Independent Database Querying (on and off the Web). *Proc. Of IDEAS2000*, 2000.

[MV00b] Ute Masermann and Gottfried Vossen. Design and Implementation of a Novel Approach to Keyword Searching in Relational Databases. *ADBIS-DASFAA Symposium*, 2000.

[Ora01] http://technet.oracle.com/products/text/content.html. 2001.

[Ple81] J. Plesn'ik. A bound for the Steiner tree problem in graphs. *Math. Slovaca 31*, pages 155–163, 1981.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Record*, 29(2):249–260, 2000.

[Sal89] Gerard Salton. Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer. *Addison Wesley*, 1989.

[Sel88] Timos K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.

[TWW+00] Jason T., L. Wang, Xiong Wang, Dennis Shasha, Bruce A. Shapiro, Kaizhong Zhang, Qicheng Ma, and Zasha Weinberg. An approximate search engine for structural databases. *SIGMOD*, 2000.

[Ull82] Jeffrey D. Ullman. Principles of Database Systems, 2nd Edition. *Computer Science Press*, 1982.