Query Planning for Searching Inter-dependent Deep-Web Databases

Fan Wang[1], Gagan Agrawal[1], and Ruoming Jin[2]

[1] Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
{wangfa,agrawal}@cse.ohio-state.edu
[2] Department of Computer Science, Kent State University, Kent OH 44242
jin@cs.kent.edu

Abstract. Increasingly, many data sources appear as online databases, hidden behind query forms, thus forming what is referred to as the *deep web*. It is desirable to have systems that can provide a high-level and simple interface for users to query such data sources, and can automate data retrieval from the deep web. However, such systems need to address the following challenges. First, in most cases, no single database can provide all desired data, and therefore, multiple different databases need to be queried for a given user query. Second, due to the dependencies present between the deep-web databases, certain databases must be queried before others. Third, some database may not be available at certain times because of network or hardware problems, and therefore, the query planning should be capable of dealing with unavailable databases and generating alternative plans when the optimal one is not feasible.

This paper considers query planning in the context of a deep-web integration system. We have developed a dynamic query planner to generate an efficient query order based on the database dependencies. Our query planner is able to select the *top K* query plans. We also develop cost models suitable for query planning for deep web mining. Our implementation and evaluation has been made in the context of a bioinformatics system, SNPMiner. We have compared our algorithm with a naive algorithm and the optimal algorithm. We show that for the 30 queries we used, our algorithm outperformed the naive algorithm and obtained very similar results as the optimal algorithm. Our experiments also show the scalability of our system with respect to the number of data sources involved and the number of query terms.

1 Introduction

A recent and emerging trend in data dissemination involves online databases that are hidden behind query forms, thus forming what is referred to as the *deep web* [13]. As compared to the *surface web*, where the HTML pages are static and data is stored as document files, deep web data is stored in databases. *Dynamic* HTML pages are generated only after a user submits a query by filling an online form.

The emergence of the deep-web is posing many new challenges in data integration. Standard search engines like Google are not able to crawl to these web-sites. At the same time, in many domains, manually submitting online queries to numerous query

B. Ludäscher and Nikos Mamoulis (Eds.): SSDBM 2008, LNCS 5069, pp. 24–41, 2008.

forms, keeping track of the obtained results, and combining them together is a tedious and error-prone process. Recently, there has been a lot of work on developing deep web mining systems [6, 7, 14, 15, 19, 31, 38]. Most of these systems focus on query interface integration and schema matching.

A challenge associated with deep web systems, which has not received attention so far, arises because the deep web databases within a specific domain are often not independent, i.e., the output results from one database are needed for querying another database. For a given user query, multiple databases may need to be queried in an *intelligent* order to retrieve all the information desired by a user. Thus, there is a need for techniques that can generate query plans, accounting for dependencies between the data sources, and extracting all information desired by a user.

A specific motivating scenario is as follows. In bioinformatics, Single Nucleotide Polymorphisms (SNPs), seem particularly promising for explaining the genetic contribution to complex diseases [3, 20, 34]. Because over seven million Single Nucleotide Polymorphisms (SNPs) have been reported in public databases, it is desirable to develop methods of sifting through this information. Much information that biological researchers are interested in requires a search across multiple different web databases. No single database can provide all user requested information, and the output of some databases need to be the input for querying another database.

We consider a query that asks for the amino acids occurring at the corresponding position in the orthologous gene of non-human mammals with respect to a particular gene, such as ERCC6. There is no database which takes gene name ERCC6 as input, and outputs the corresponding amino acids in the orthologous gene of non-human mammals. Instead, one needs to execute this query plan. We first need to query on one database, such as SNP500Cancer, to retrieve all SNPs located in gene ERCC6. Second, using the extracted SNP identifier, we query on SNP database, such as dbSNP, to obtain the amino acid position of the SNP. Third, we need to use a sequence database to retrieve the protein sequence of the corresponding SNP. Finally, querying on BLAST database, which is a sequence alignment database, we can obtain the amino acid at the corresponding position in the orthologous gene of non-human mammals.

From the above example, we can clearly see that for a particular query, there are multiple *sub-goals*. These sub-goals are not specified by the user query, because the user may not be familiar with details of the biological databases. The query planner must be able to figure out the *sub-goals*. Furthermore, we can note the strong dependencies between those databases, which constraint the query planning process.

This paper considers query planning in the context of a deep-web integration system. The system is designed to support a very simple and easy to use query interface, where each query comprises a *query key term* and a set of *query target terms* that the user is interested in. The query key term is a name, and the query target terms capture the properties or the kind of information that is desired for this name. We do not need the user to provide us with a formal predicate-like query. In the context of such a system, we develop a dynamic query planner to generate an efficient query order based on the deep web database dependencies. Our query planner is able to select the *top* K query plans. This ensures that when the most efficient query plan is not feasible, for examples, because a database is not available, there are other plans possible.

To summarize, this paper makes the following contributions:

1. We formulate the query planning and optimization problem for deep web databases with dependencies.
2. We design and implement a dynamic query planner to generate the top K query plans based on the user query and database dependencies. This strategy provides alternative plans when the most efficient one is not feasible due to the non-availability of a database.
3. We support query planning for a user-friendly system that requires the user to only include query key terms and a set of target terms of interest. Database schemas are input by an administrator or designer.
4. We develop cost models suitable for query planning for deep web mining.
5. We present an integrated approximate planning algorithm with approximation ratio of $1/2$.
6. We integrate our query planner with a deep web mining tool SNPMiner [37] to develop a domain specific deep web mining system.
7. We evaluate our dynamic query planning algorithm with two other algorithms and show that our algorithm can achieve optimal results for most queries, and furthermore, our system has very good scalability.

The rest of the paper is organized as follows. In Section 2, we formulate the dynamic query planning problem. We describe the details of our dynamic query planner in Section 3. In Section 4, we evaluate the system. We compare our work with related efforts in Section 5 and conclude in Section 6.

2 Problem Formulation

The deep web integration system we target provides a fixed set of candidate terms which can be queried on. These terms are referred to as the *Query Target Terms*. A user selects a subset of the allowable *Query Target Terms* and in addition, specifies a *Query Key Term*. Query target terms specify what type of information the user wants to know about the query key term. From the example in Section 1, we know that for a single query of this nature, several pieces of information may need to be extracted from various databases. Furthermore, there are dependencies between different databases, i.e. information gained from one source may be required to query another source. Our goal is to have a *query planning* strategy that can provide us an efficient and correct query plan to query the relevant databases.

We have designed a dynamic query planner which can generate a set of *Top* K query plans. The query plan with shorter *length*, i.e. the number of databases searched, higher coverage of user request terms, and higher user preference, are considered to have a higher priority. By generating K query plans, there can be back-up plans when the best one is not feasible, for example, because of unavailability of a database.

Formally, the problem we consider can be stated as follows. We are given a universal set $T = \{t_1, t_2, \ldots, t_n\}$, where each t_i is a term that can be requested by a user. We are also given a subset $T' = \{t'_1, t'_2, \ldots, t'_m\}$, $t'_i \in T$, of terms that are actually requested by the user for a given query. We also have a set $D = \{D_1, D_2, \ldots, D_m\}$, where each

D_i is a deep web database, and each D_i covers a set of terms $E_i = \{e_i^1, e_i^2, \ldots, e_i^k\}$, and E_i is a subset of T. Furthermore, each database D_i requires a set of elements $\{r_i^1, r_i^2, \ldots, r_i^k\}$ before it can be queried, where $r_i^j \in T$.

Our goal is to find a query order of the databases $D^* = \{D_1, D_2, \ldots, D_k\}$, which can cover the set T' with the maximal *benefit* and also makes k as small as possible. The benefit is based on a cost function that we can choose. We call it a dynamic query planning problem, because the query order should be selected based on the user specified target terms, and cannot be fixed by the integration system. This is a variant of the famous weighted set cover problem, and can be easily proven as NP-Complete [9].

2.1 Production System Formulation

For discussing our algorithm, it is useful to view the query planning problem as a *production system*. A production system is a model of computation that has proved to be particularly useful in AI, both for implementing search algorithms and for modeling human problem solving [27]. A production system can be represented by four elements, which are a *Working Memory*, a *Target Space*, a set of *Production Rules*, and a *Recognize-Act* control cycle. The working memory contains a description of the current state in a reasoning process. The target space is the description of the aim. If the *working memory* becomes a superset of the *target space*, the problem solving procedure is completed. A production rule is a *condition-action* pair. The *condition* part determines whether the rule can be applied. The *action* part defines the associated problem-solving step. The *working memory* is initialized with the beginning problem description. The current state is matched against the conditions of the production rules. When a production rule is fired, its action is performed, and the working memory is changed accordingly. The process terminates when the content of the working memory becomes a superset of the target state, or no more rules can be fired.

We map our query planning problem into the four elements of a production system as follows. The working memory is comprised of all the data which has already been extracted. Our query plan is generated step by step, and when a database is added into our query plan, the data that can be obtained from this database is considered as stored in the working memory. Initially, the working memory is just the Query Key Term. The target state is a subset of the Query Target Terms selected by the user.

Each online database has one or more underlying query schema. Those schemas specify what the input of the online query form of the database is, and what data can be extracted from the database by using the input terms. The production rules of our system are the database schemas. Note that one database may have multiple schemas. In this case, each schema carries different input elements to retrieve different output results. The database schemas are provided by deep web data source providers and/or a developer creating the query interface.

The terms in working memory are matched against the necessary input set of each production rule. Appropriate rule will be fired according to our rule selection strategy, which will be introduced in Section 3.3. We consider the corresponding database as queried and the output component of the fired rule is added to the working memory. We mark the selected rules as visited to avoid re-visiting the same rule. If either of the following two cases holds, one complete query plan would have been generated. In the

first case, the working memory has covered all the elements in the target space, which means that all user requested Query Target Terms have been found. In the second case, there are still some terms in the target state have not been covered by the working memory, but no unvisited rules can cover any more elements in the target space. This means that it is impossible to retrieve all the request terms by using current set of available databases. This normally occurs when some databases are unavailable.

3 Query Planning Approach and Algorithm

Our approach for addressing the problem stated in the previous section is as follows. We first introduce a data structure, *dependency graph*, to capture the database dependencies. Our algorithm is based on this data-structure. Towards the end of this section, we describe the cost or *benefit* model that we use.

3.1 Dependency Graph

As we stated earlier, there are dependencies between online databases. If we want to query database D, we have to query on some other databases in order to extract the necessary input elements of D first. We use the production rule representation of the databases to identify the dependecies between the databases and build a dependency graph of databases to capture the relationship between databases.

Formally, there is a dependency relation DR, $\prec_{DR} \subset 2^D \times D$, where 2^D is the power set of D. If $\{D_i, D_{i+1}, \ldots, D_{i+m}\} \prec_{DR} D_j$, we have to query on data source $D_i, D_{i+1}, \ldots, D_{i+m}$ first in order to obtain the necessary input elements for querying on the data source D_j. Note that there could be multiple combinations of databases that can provide input required for querying a given database.

We use *hypergraph* to represent the dependency relationship. A hypergraph consists of a set of nodes N and a set of hyperarcs. The set of hyperarcs is defined by ordered pairs in which the first element of the pair is a subset of N and the second element is a single node from N. The first element of the pair is called the parent set, and the second element of the pair is called the descendant. If the parent set is not singleton, the elements of the set of parents are called *AND* nodes. In our dependency graph, the nodes are online databases, and hyperarcs represent the dependencies between databases. For a particular hyperarc, the parent nodes of the pair are the databases which must be queried first in order to continue the query on the database represented by the descendent node of the pair.

The dependency graph is constructed using the production rules of each online database. For two databases D_i and D_j, suppose D_i has a set of production rules $R_i = \{r_{i1}, r_{i2}, \ldots, r_{in}\}$ and D_j has a set of production rules $R_j = \{r_{j1}, r_{j2}, \ldots, r_{jm}\}$. If any rule in R_i has an output set which can fully cover any of the rules' input set in R_j, we build an edge between D_i and D_j. In another case, if any rule in R_i has an output set which partially covers any of the rules' input set in R_j, we scan the rules of other databases to find a partner set of databases for D_i together with which can fully cover any of the rules' input set in R_j. If the partner set exists, we build a hyperarc from D_i

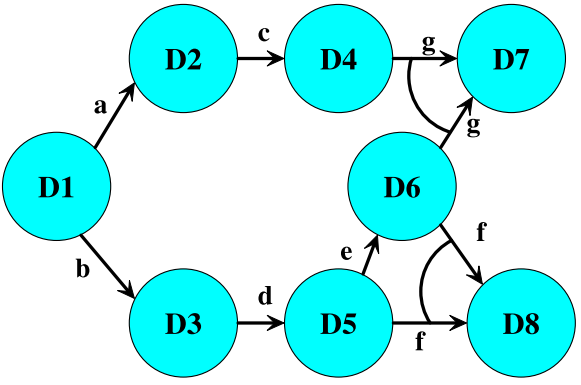


Fig. 1. Dependency Graph Example

and its partner set databases to D_j. If the production rules of a database are updated, our dependency graph can also be updated accordingly. Figure 1 shows an example of hypergraph.

In Figure 1, there are 8 nodes in the hypergraph, and 7 hyperarcs. The first hyperarc is denoted by a. The first element (parent) of the ordered pair of a is node D1, and the second element (descendant) of the pair is node D2. This hyperarc implies that after querying on D1, we can obtain the input elements needed for querying D2. Hyperarcs b to e are similar to hyperarc a. In hyperarc f, the arc connecting the two edges of f shows that this hyperarc is a *2-connector*. The first element of f is a set comprising of D5 and D6, and the second element is D8. This shows in order to query on D8, we need to first query on D5 and D6. Hyperarc g has the same structure as f.

For a node D, the *neighbors* of D are the nodes which have an edge coming from D. The partners of D are the nodes which is connected with D by a hyperarc to a common descendent. For example, in Figure 1, $D5$ has a partner $D6$, as $D5$ and $D6$ connect $D8$.

3.2 Query Planning Algorithm

We now introduce our algorithm. Initially, we define several new terms.

Due to the existence of database dependencies, some databases can be more important because it can link us to other important databases. In Figure 1, suppose we are on node $D1$, and user requested terms can only be obtained by the database $D8$. Using the query key term, we cannot query on $D8$ at the beginning. We must first query on other databases, say $D3$, to gain additional information in order to follow the database dependencies and finally query on $D8$. We call databases $D2$, $D3$, $D4$, $D5$ and $D6$ *hidden nodes*.

In order to determine the hidden nodes, we need to come up with a strategy to find all reachable nodes from a starting node in the dependency graph. This can be done by adapting the breath-first search method to a hypergraph. We use the algorithm *Find_Reachable_Node(DG,s)* to find all reachable nodes from a starting node s in the dependency graph DG. We show the sketch of this algorithm in Algorithm 3.1. In the

algorithm, $Q1$ is a queue which stores all reachable nodes from starting node s, $Q2$ stores all reachable nodes from s with the help of its partners, and $PS(t)$ returns the partner set of t.

Algorithm 3.1: FindReachableNodes(DG, s)

```
Initialize two queues Q1 and Q2
Add s to Q1, and mark s as visited
while Q1 is not empty
  Dequeue the first element in Q1, name it as t
  foreach n which is a neighbor of t
    if n ∈ unvisited and PS(t) = Φ and rules match
      Add n to Q1 and mark n as visited
      else if n ∈ unvisited and PS(t) ≠ Φ
      Add n to Q2
  while Q2 is not empty
    foreach n ∈ Q2
      Extract the partner set PS of n
      Denote each partner of n as p
      if p ∈ Q1
        foreach p of n and rules match
          Add n to Q1, and remove n from Q2
          Mark n as visited
return (Q1)
```

Next, we introduce a new concept, *Database Necessity*. Each production rule is associated with a set of terms which can be extracted by executing the rule. Some terms can only be provided by one database, while other terms can be provided by multiple databases. If a requested term can only be provided by a single rule, that rule should have a higher priority to be executed. Conversely, if the term can be provided by multiple rules, a lower priority can be assigned to this rule. Based on this idea, each term is associated with a *Database Necessity* value. Formally, for a term t, if K databases can provide it, the database necessity value for t is $\frac{1}{K}$.

As part of our algorithm, we need to make hidden rules partially *visible on the surface*. A hidden but potentially useful rule has the following two properties: (1) It must be executed in order to extract all user requested terms. (2) The necessary input elements of it are hidden, i.e. either they are not in the Query Target Terms or they can only be extracted by rules located in the hidden layer.

We make the hidden rules visible in a bottom-up manner as follows. We scan all the terms in the user provided Query Target Terms, i.e. the initial Target Space. If there is some term in the initial target space with database necessity value of 1, which means only one production rule, say R, can provide this term. This rule R is a hidden rule which must be fired. In order to make rule R visible, we add the necessary input elements of R into the target space to enlarge the target space. Then, we re-scan the newly enlarged target space to find new hidden rules and enlarge the target space further. This procedure continues until there are no more hidden rules of this kind.

Another important issue in our algorithm is the ability to prune similar query plans. The dynamic query planner can generate the top K query plans. When one query plan is generated, the algorithm will trace back from the current point to generate another query plan. It is highly possible that two generated query plans QP_1 and QP_2 use the same set of databases, but differ in the order of querying two or more databases which do not have any dependencies. In this case, we will consider the two query plan QP_1 and QP_2 as the same, and the latter one will be deleted.

Algorithm 3.2: Find_TopK_Query_Plans(PR, WS, TS)

```
while enlargeable(WS)
  Enlarge WS
Initialize queue Q and P
while size(Q) ≤ K
  if (∃e ∈ TS and e ∉ WS)
   and (∃r ∈ PR and ∃o ∈ O(r) and o ∈ TS)
    Find candidate rule set CR
    foreach r ∈ CR
      Compute benefit score according to benefit model
    Select r_opt, the rule with the highest benefit
    if !prunable(P, r_opt)
      while r_opt ≠ null and (∃e ∈ TS and e ∉ WS)
       and (∃r ∈ PR and ∃o ∈ O(r) and o ∈ TS)
        Add r_opt to P, and update WS
        Select next r_opt
      else Empty queue P
    else Add P to Q and re-order Q
  if size(P) > 0
    Remove the last rule of P, update WS, trace back
return (Q)
```

Main Algorithm. Our dynamic query planning algorithm takes the Query Target Terms and Query Key Term, and dynamically generates K query plans which cover as many request terms as possible. At the beginning, we enlarge user provided target space to visualize hidden rules and obtain the enlarged target space. We take the Query Key Term as the initial working memory. Then, the production system begins the recognize-act procedure. Each iteration the system selects an appropriate rule according to a benefit model and updates the current working memory. This procedure terminates when all terms in the target space are covered or no more rules can be fired.

Now, we assume our benefit model can select the *best* rule according to the current working memory and our goal. We will introduce our benefit model in detail in Section 3.3. Algorithm 3.2 shows the sketch of the planning algorithm. In the algorithm, PR is the set of production rules, WS is the working memory, and TS is the target space. Q is a queue to store the top K query plans, and P is a queue to store the rules along one query plan. $O(r)$ returns the output elements of a rule r. $prunable()$ is a function to test whether a candidate query plan can be pruned as we discussed earlier.

Our dynamic query planning algorithm is a greedy algorithm which selects the production rule with the local maximal benefit according to the benefit model. Each greedy algorithm has an approximation ratio which measures the performance of the algorithm. We use $|R|$ to represent the cardinality of the collection of rules R, i.e. the total number of production rules. We have the following result:

Theorem 1. *The approximation algorithm introduced in Algorithm 3.2 has an approximation ratio of* $\frac{|R|+1}{2|R|}$.

The proof is omitted for lack for space.

3.3 Benefit Model

A very important issue in a production system is *rule selection*, i.e., which rule should be executed. We have designed a benefit model to select an appropriate rule at each iteration of the recognize-act cycle. In the algorithm presented earlier in this section, at each step, each rule is scanned and all the rules which can be fired are put into a set called the *candidate rule set*. Then, we compute a benefit score for each of the candidate rules.

We have used four metrics for rule selection, which are Database Availability (DA), Data Coverage (DC), User Preference (UP), and Potential Importance (PI).

Database Availability: A production rule R can be executed if the corresponding database is available. In our implementation, for each rule, we send a message to the database to test the availability of the database. If the database is not available, we just ignore this rule for the current iteration.

Data Coverage: Data coverage measures the percentage of required data that can be provided by a particular rule. Given a rule R_k, the target state TS, and $k-1$ rules $R_1, R_2, \ldots, R_{k-1}$ that have already been selected, we want to compute the data coverage of the current rule R_k with respect to TS. We use the number of Query Target Terms in TS which are also covered by the rule R_k, but have not been extracted by previous rules for this purpose.

User Preference: Some terms can be extracted from multiple databases, and domain users may have preference for certain databases for a particular term. We can assign a user preference value for each term with respect to databases and incorporate user preference into the benefit function. Consider a particular term t, which can be obtained from r databases $D_1, D_2, \ldots, D_r$. A number between 0 and 1 should be assigned to t for each of the r databases as the preference value, such that the r preference values sum up to 1. If t can only be obtained from a single database D, the preference value of t with respect to D is 1 and is 0 for all other databases. The user preference values should be given by a domain expert.

Suppose we are examining the production rule R, which is associated with the database D. The following k terms $UF_1, UF_2, \ldots, UF_k$ have not been found. For each term UF_i, the user preference with respect to database D is UP_i. We use the database necessity value of each term (DN_i for term UF_i) as the weight of its user preference

and we compute the weighted sum of all unfound terms as the user preference value of the rule, i.e. the user preference of R is $\sum_{i=1}^{k} DN_i * UP_i$.

Potential Importance: Because of database dependencies, some databases can be more important due to its linking to other important databases. Figure 1 shows an example. In the above case, suppose $D2$ and $D3$ have the same data coverage and user preference. Obviously, $D3$ is potentially more important because $D3$ can help us link to our final target $D8$. As a result, the $D3$ should be assigned a larger benefit value. Based on the above idea, we incorporate potential importance to our benefit function.

Suppose we are considering production rule corresponding to the database D. By using Algorithm 3.1, we find a set of databases $D_{reachable} = \{D_1, D_2, \ldots, D_m\}$, which can be queried by using the data extracted from database D exclusively. We have k term which have not been found, denoted by $UF_1, UF_2, \ldots, UF_k$. For term UF_i, its database necessity value is DN_i, which means the term UF_i can be obtained by $\frac{1}{DN_i}$ number of databases and we denote this set of databases as $NecessaryD_i$. We want to know the number of *Necessary Databases* of UF_i which can be reached by the current rule R. We count the number of databases in $NecessaryD_i$, which are also in the set $D_{reachable}$, i.e. we compute the cardinality of the set $\{d | d \in NecessaryD_i, d \in D_{reachable}\}$. Suppose the cardinality is r_i for term UF_i. The potential importance for UF_i with respect to rule R and corresponding database D is $\frac{r_i * \frac{1}{DN_i}}{|D_{reachable}|} = \frac{r_i}{m * DN_i}$. Finally, the potential importance for the rule R is

$$\sum_{i=1}^{k} \frac{r_i}{m * DN_i}$$

For each candidate rule, a benefit score is computed according to the three metrics, data coverage, user preference and potential importance. The value of the three metrics are closely related to the database necessity values of all unfound terms when a rule is being examined, as a result, if a rule is considered as a candidate multiple times, each time the benefit score must be different, because each time the set of unfound terms is different. As a result, the benefit score of a production rule is dynamically related to the current working space of the production system.

The benefit function of a rule R with respect the current working space WS can be represented as follows:

$$BF(R, WS) = DC * \alpha + UP * \beta + PI * \gamma, \alpha + \beta + \gamma = 1$$

There are three parameters α, β and γ associated with each metric term. These three parameters scale the relative importance of the three metric terms.

3.4 Discussion: System Extendibility

Extendibility is an important issue for any deep web mining system, as new data sources can emerge often. We now briefly describe how a new data source can be integrated with our system. First, we need to represent the database query schemas of the new data source into the form of production rules. Then, a domain expert assigns or changes user

preference values for the terms appearing in the newly integrated data sources. We have developed simple algorithms for automatically integrating the new data source into the Dependency Graph and updating the database necessity values. The algorithms proposed are scalable to larger numbers of databases. Furthermore, because the design of our dependency graph and query planning algorithm is based on the inherent characteristics of deep web data sources, such as database dependencies and database schemas, our system is independent of the application domain, i.e., the system can be applied on any domains of application.

4 Performance

This section describes the experiments we conducted to evaluate our algorithm. We ran 30 queries and compared the performance of our algorithm with two other algorithms.

4.1 Experiment Setup

Our evaluation is based on the SNPMiner system [37]. This system integrates the following biological databases: dbSNP[1], Entrez Gene and Protein[2], BLAST[3], SNP500Cancer[4], SeattleSNPs[5], SIFT[6], and BIND[7]. SNPMiner System provides an interface by which users can specify query key terms and query target terms. We use some heuristics to map user requested keywords to appropriate databases. SNPMiner uses Apache Tomcat 6.x to support a web server. After a query plan is executed, all results are returned in the form of HTML files. We have a web page parser to extract relevant data from the files and tabulate the data.

We created 30 queries for our evaluation. Among these 30 queries, 10 are real queries specified by a domain expert we have collaborated with. The remaining 20 queries were generated by randomly selecting query keywords. We also vary the number of terms in each query in order to evaluate the scalability of our algorithm. Table 1 summarizes the statistics for the 30 queries.

Table 1. Experimental Query Statistics

Query ID	Number of Terms
1-8	2-5
9-16	8-12
17-24	17-23
25-28	27-33
29,30	37-43

[1] http://www.ncbi.nlm.nih.gov/projects/SNP
[2] http://www.ncbi.nlm.nih.gov/entrez
[3] http://www.ncbi.nlm.nih.gov/blast/index.shtml
[4] http://snp500cancer.nci.nih.gov/home_1.cfm
[5] http://pga.gs.washington.edu/
[6] http://blocks.fhcrc.org/sift/SIFT.html
[7] http://www.bind.ca

Our evaluation has three parts. First, we compare our production rule algorithm with two other algorithms. Second, we show that enlarging the target space improves the performance of our system significantly. Finally, we evaluate the scalability of our system with respect to the number of databases and the number of query terms. In all our experiments, the three scaling parameters are set as follows: $\alpha = 0.5$, $\beta = 0.3$ and $\gamma = 0.2$.

In comparing planning algorithms or evaluating the impact of an optimization, we use two metrics, which are the *number of databases involved in the query plan* and the *actual execution time for the query plan*. We consider a query plan to be good if it can cover all user requested terms using as few databases as possible. A query plan that involves more databases tends to query redundant databases, and cause additional system workload.

4.2 Comparison of Three Planning Algorithms

We compare our Production Rule Algorithm (PRA) with two other algorithms, which are the Naive Algorithm (NA) and the Optimal Algorithm (OA).

Naive Algorithm: As the name suggests, this algorithm does query planning in a naive way. The algorithm selects all production rules which can be queried at each round, until all keywords are covered. This algorithm can quickly find a query plan, but the query plan is likely to have a very low score and a long execution time.

Optimal Algorithm: This algorithm searches the entire space to find the optimal query plan. Because we only had 8 databases for our experiments, we could manually determine the optimal query plan for each query. Such a plan is determined based on the number of databases involved in the query plan and the expected response time of the databases involved. This means that the optimal query plan has the smallest *estimated* execution time, though the measured execution time may not necessarily be the lowest of all plans.

In Figure 2, sub-figures (1a) and (1b) show the the comparison between PRA and NA. In sub-figure (1a), the diamonds are the ratios between the execution time of the query plans generated by PRA and NA, annotated as *ETRatio*. We can see that all diamonds are located below the $ratio = 1$ line, which implies that for each of the 30 queries, the query plan generated by production rule algorithm has a lesser execution time than that of the plan generated by naive algorithm. In the sub-figure (1b), the rectangles are the ratios of the number of databases involved in the query plan generated by PRA and NA, denoted as *DRatio*. We observe that the same pattern, i.e. the query plans generated by the production rule algorithm use fewer data sources.

Sub-figures (2a) and (2b) show the the comparison between PRA and OA. From the sub-figure (2a), we can observe that all the diamonds are distributed closely around the $ratio = 1$ line. This shows that in terms of the execution time of generated query plans, the production rule algorithm has close to the optimal performance. We also observe from the sub-figure (2b) that in terms of the number of databases involved in query plans, the production rule algorithm obtains the optimal result, with an exception of query 11. We examined the query plans generated by PRA, and found that most of the query plans are exactly the same as the optimal query plans. For other cases, we note

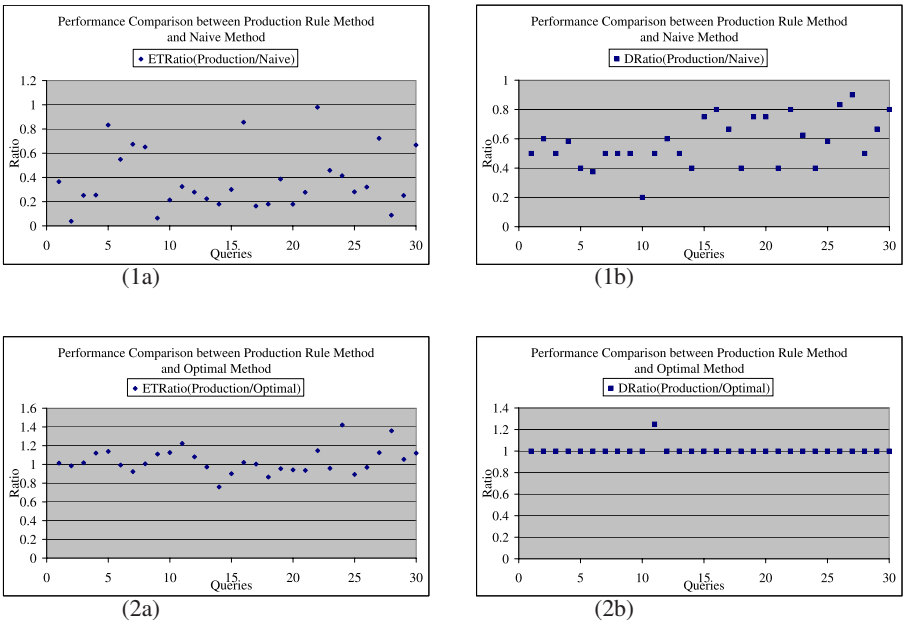


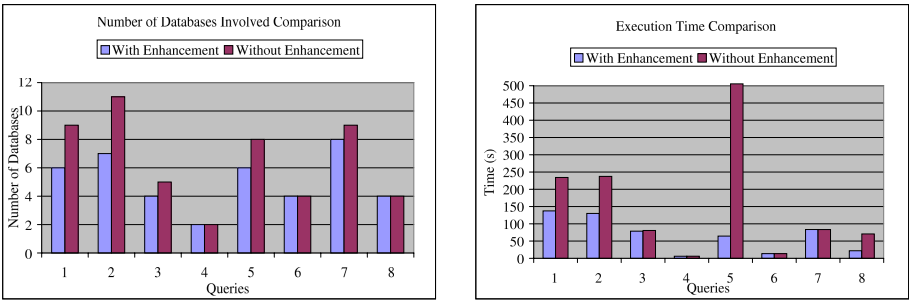
Fig. 2. Comparison among PRA, NA and OA: (1a) Comparison between PRA and NA on Plan Execution Time; (1b) Comparison between PRA and NA on Plan Length;(2a) Comparison between PRA and OA on Plan Execution Time;(2b) Comparison between PRA and OA on Plan Length

that the optimal algorithm uses some databases with lower response time. However, this did not necessarily result in lower actual execution time. We can see from the sub-figure (2a) that some of the execution time with PRA are actually smaller than the execution times of the plans generated by the optimal algorithm.

4.3 Impact of Enlarging Target Space

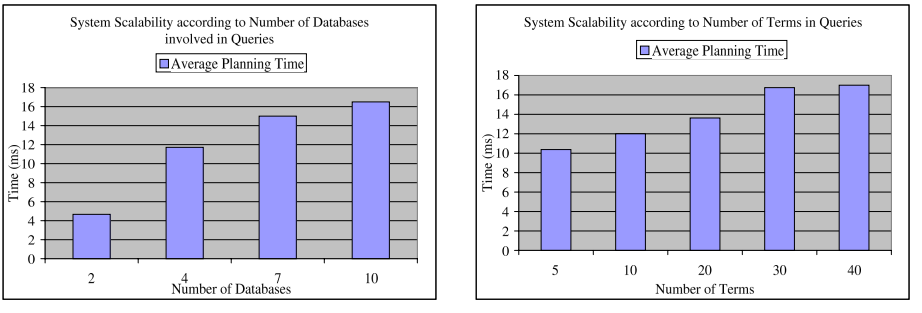
In this experiment, we compare the number of databases involved and the execution time for different query plans generated using the system without enlarging target space and the system with enlarged target space. We select 8 queries which contains many terms with database necessity value smaller than 1, because these query plan results can better show the usefulness of enlarging the target space. The results are shown in Figure 3.

We have the following observations. From the sub-figure (a) in Figure 3, we can observe that the number of databases involved for most of the query plans is much shorter for the enhanced system than that of the system without enhancement. From the sub-figure (b) in Figure 3, we can also observe that the execution time reduces very significantly for the enhanced system. The above results show that enlarging target space can effectively improve our system to generate query plans with fewer databases and less execution time.



(a) (b)

Fig. 3. System Enhancement Test: (a) Comparison of Number of Databases Involved; (b) Comparison of Execution Time



(a) (b)

Fig. 4. System Scalability Test: (a) System Scalability with respect to Number of Databases Involved; (b) System Scalability with respect to Number of Terms in Queries

4.4 Scalability of Production Rule Algorithm

Our last experiment evaluated how the query planning time scales with increasing number of databases and query terms. From Figure 4, in sub-figure (a), we can observe that in terms of the average planning time, there is a sharp increase in going from 2 data sources to 4 data sources. Then, the planning time increases only moderately with respect to the increase in the number of data sources. In the sub-figure (b), we can see that the average planning time increases very slowly with the increase in the number of terms in the queries. This shows that our system has good scalability.

5 Related Work

We now compare our work with existing work on query planning, deep web mining, and keyword search on relational databases.

Query Planning: There are a number of research efforts on query planning. Raschid and co-workers have developed a navigational-based query planning strategy for mining

biological data sources [5,21,22,23,28,36]. They build a source graph representing integrated biological databases and an object graph representing biological objects in each database. The navigational links (hyperlink) between the database objects are assumed to be pre-fetched. Extending their work [23, 28], they allowed each physical link to carry a semantic meaning to enhance their algorithm. The key differences in our work are as follows. First, we focus on deep web database dependencies, not the physical links between database objects. Further, in their work, a user query needs to specify source and target databases.

A lot of work has been done in SQL-based query planning [1, 12, 18, 26, 32]. In [18], new SQL operators were introduced to reduce repetitive and unnecessary computations in query planning. In [12, 26], a set of pre-defined plans were represented in the form of grammar production rules. For a SQL query, the algorithm first built plans to access individual tables, and then repeatedly refered grammar rules to join plans that were generated earlier. Other work has focused on query planning using database views and multiple databases sharing the same relational schema [1, 32].

Much work on query planning is based on the well known *Bucket Algorithm* [10, 11, 17, 24, 25, 29, 30]. In the above work, they assume that the user query specifies the databases or relations need to be queried, and the task of the work is to find a query order among the specified relations or databases. Based on user specified relations or sub-goals, a bucket is built containing all the databases which can answer the corresponding sub-goal. But in our work, the user query only contains keyword and will not specify any databases or relations of interest. Our system selects the best data sources automatically, i.e. our system figure out sub-goals by itself. At the same time, query planning is performed.

In [35], a query planning algorithm minimizes the query's total running time by optimally exploits parallelism among web services. The main difference between our work and theirs is, they assume that one attribute can only be provided by exactly one data source which is a unrealistic assumption in real application, but we allow the present of data redundancy.

Deep Web Mining: Lately, there has been a lot of work on mining useful information from the deep web [6, 7, 14, 15, 19, 31, 38]. In [19], a database selection algorithm based on attribute co-occurrence graph was proposed. In [31], Nie *et al.* proposed an object-level vertical searching mechanism to handle the disadvantages of document-level retrieval. QUIC [38] was a mining system supporting imprecise queries over incomplete autonomous databases. In [14, 7, 6], Chang *et al* proposed an E-commerce domain deep web mining tool MetaQuerier. MetaQuerier translated user query into several local queries by schema matching. WISE-Integrator [15] was another deep web mining tool similar to MetaQuerier. The key difference in our work is that none of the above systems consider database dependencies.

Keyword Search on Relational Databases: Recently, providing keyword based search over relational databases has attracted a lot of attention [2,4,16,33]. The major technical issue here is to efficiently search several keywords which co-occur in the same row, in a table obtained by joining multiple tables or even databases together, and rank them

based on different measures. In addition, the keywords may appear in any attribute or column. This is very different from the problem studied in this paper.

Select-Project-Join Query Optimization: There has been extensive work in query optimization, especially SPJ type query optimization since the early 1970s [8]. A query optimizer needs to generate an efficient execution plan for the given SQL query from a space of possible plans based on a cost estimation technique which is used to measure the cost of each plan in the search space. Our work has some similarities with the above research efforts in that we both do selection as earlier as possible. Two major differences between our work and SPJ query optimization are as follows. First, in traditional query optimization, any join-order is allowed, but for our work, due to deep web properties, the allowable join operations are restricted. Second, in traditional databases, redundant columns seldom occur, so it is impossible to have options to take one project or column from several alternative databases, but redundant data exists in our deep web databases, and we can take different paths. As pointed out above, our problem is different from traditional SPJ query and new techniques are needed.

6 Conclusion

In this paper, we formulated and solved the query planning and optimization problem for deep web databases with dependencies. We have developed a dynamic query planner with an approximation algorithm with a provable approximation ratio of $1/2$. We have also developed cost models to guide the planner. The query planner automatically selects best sub-goals on-the-fly. The K query plans generated by the planner can provide alternative plans when the optimal one is not feasible. Our experiments show that the cost model for query planning is effective. Despite using an approximate algorithm, our planning algorithm outperforms the naive planning algorithm, and obtains the optimal query plans for most experimental queries in terms of both number of databases involved and actual execution time. We also show that our system has good scalability.

References

1. Abiteboul, S., Garcia-Molina, H., Papakonstantinou, Y., Yerneni, R.: Fusion queries over internet databases (1997)
2. Agrawal, S., Chaudhuri, S., Das, G.: Dbxplorer: A system for keyword-based search over relational databases. In: Proceedings of the 18th International Conference on Data Engineering, pp. 5–16 (2002)
3. Brookes, A.J.: The essence of snps. Gene. 234, 177–186 (1999)
4. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using banks. In: Proceedings of the 18th International Conference on Data Engineering, pp. 431–440 (2002)
5. Bleiholder, J., Khuller, S., Naumann, F., Raschid, L., Wu, Y.: Query planning in the presence of overlapping sources. In: Proceedings of the 10th International Conference on Extending Database Technology, pp. 811–828 (2006)
6. Chang, K., He, B., Zhang, Z.: Toward large scale integration: Building a metaquerier over databases on the web (2005)

7. Chang, K.C.-C., Cho, J.: Accessing the web: From search to integration. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of Data, pp. 804–805 (2006)
8. Chaudhuri, S.: An overview of query optimization in relational systems. In: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pp. 34–43 (1998)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press, Cambridge (2001)
10. Doan, A., Halevy, A.: Efficiently ordering query plans for data integration. In: Proceedings of the 18th International Conference on Data Engineering, p. 393 (2002)
11. Florescu, D., Levy, A., Manolescu, I.: Query optimization in the presence of limited access patterns. In: Proceedings of the 1999 ACM SIGMOD international conference on Management of Data, pp. 311–322 (1999)
12. Haas, L.M., Kossmann, D., Wimmers, E.L., Yang, J.: Optimizing queries across diverse data sources. In: Proceedings of the 23rd International Conference on Very Large Databases, pp. 276–285 (1997)
13. He, B., Patel, M., Zhang, Z., Chang, K.C.-C.: Accessing the deep web: A survey. Communications of ACM 50, 94–101 (2007)
14. He, B., Zhang, Z., Chang, K.C.-C.: Knocking the door to the deep web: Integrating web query interfaces. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of Data, pp. 913–914 (2004)
15. He, H., Meng, W., Yu, C., Wu, Z.: Automatic integration of web search interfaces with wise_integrator. The international Journal on Very Large Data Bases 12, 256–273 (2004)
16. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient ir-style keyword search over relational databases. In: Proceedings of the 29th international conference on Very large data bases, pp. 850–861 (2003)
17. Ives, A.G., Florescu, D., Friedman, M., Levy, A.: An adaptive query execution system for data integration. In: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, pp. 299–310 (1999)
18. Jin, R., Agrawal, G.: A systematic approach for optimizing complex mining tasks on multiple databases. In: Proceedings of the 22nd International Conference on Data Engineering, p. 17 (2006)
19. Kabra, G., Li, C., Chang, K.C.-C.: Query routing: Finding ways in the maze of the deep web. In: Proceedings of the 2005 International Workshop on Challenges in Web Information Retrieval and Integration, pp. 64–73 (2005)
20. Lohnmueller, K.E., Pearce, C.L., Pike, M., Lander, E.S., Hirschhorn, J.N.: Meta-analysis of genetic association studies supports a contribution of common variants to susceptibility to common disease. Nature Genet. 33, 177–182 (2003)
21. Lacroix, Z., Parekh, K., Vidal, M.-E., Cardenas, M., Marquez, N., Raschid, L.: Bionavigation: Using ontologies to express meaningful navigational queries over biological resources. In: Proceedings of the 2005 IEEE Computational Systems Bioinformatics Conference Workshops, pp. 137–138 (2005)
22. Lacroix, Z., Raschid, L., Vidal, M.-E.: Efficient techniques to explore and rank paths in life science data sources. In: Proceedings of the 1st International Workshop on Data Integration in the Life Sciences, pp. 187–202 (2004)
23. Lacroix, Z., Raschid, L., Vidal, M.E.: Semantic model to integrate biological resources. In: Proceedings of the 22nd International Conference on Data Engineering Workshops, p. 63 (2006)
24. Leser, U., Naumann, F.: Query planning with information quality bounds. In: Proceedings of the 4th International Conference on Flexible Query Answering, pp. 85–94 (2000)

25. Levy, A.Y., Rajaraman, A., Ordille, J.J.: Querying heterogeneous information sources using source descriptions. In: Proceedings of the 22nd International Conference on Very Large Databases, pp. 251–262 (1996)
26. Lohman, G.M.: Grammar-like functional rules for representing query optimization alternatives. In: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, pp. 18–27 (1988)
27. Luger, G.F.: Artificial intelligence: Structure and Strategies for Complex Problem Solving, 5th edn. Addison-Wesley, Reading (2005)
28. Mihaila, G., Raschid, L., Naumann, F., Vidal, M.E.: A data model and query language to explore enhanced links and paths in life science sources. In: Proceedings of the eighth International Workshop on Web and Databases, pp. 133–138 (2005)
29. Naumann, F., Leser, U., Freytag, J.C.: Quality-driven integration of heterogeneous information systems. In: Proceedings of the 25th International Conference on Very Large Data Bases, pp. 447–458 (1999)
30. Nie, Z., Kambhampati, S.: Joint optimization of cost and coverage of information gathering plans, asu cse tr 01-002. computer science and engg. arizona state university
31. Nie, Z., Wen, J.-R., Ma, W.-Y.: Object-level vertical search. In: Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research, pp. 235–246 (2007)
32. Pottinger, R., Levy, A.: A scalable algorithm for answering queries using views. The international Journal on Very Large Data Bases 10, 182–198 (2001)
33. Sayyadian, M., LeKhac, H., Doan, A., Gravano, L.: Efficient keyword search across heterogeneous relational databases. In: Proceedings of the 23rd International Conference on Data Engineering, pp. 346–355 (2007)
34. Chanock, S.: Candidate genes and single nucleotide polymorphisms (snps) in the study of human disease. Disease Markers 19, 89–98 (2001)
35. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query optimization over web services. In: Proceedings of the 32nd international conference on Very Large Data Bases, pp. 355–366 (2006)
36. Vidal, M.-E., Raschid, L., Mestre, J.: Challenges in selecting paths for navigational queries: Trade-off of benefit of path versus cost of plan. In: Proceedings of the 7th International Workshop on the Web and Databases, pp. 61–66 (2004)
37. Wang, F., Agrawal, G., Jin, R., Piontkivska, H.: Snpminer: A domain-specific deep web mining tool. In: Proceedings of the 7th IEEE International Conference on Bioinformatics and Bioengineering, pp. 192–199 (2007)
38. Wolf, G., Khatri, H., Chen, Y., Kambhampati, S.: Quic: A system for handling imprecision and incompleteness in autonomous databases. In: Proceedings of the Third Biennial Conference on Innovative Data Systems Research, pp. 263–268 (2007)