

From: Coward, Jeffery

Sent: 10/29/2013 4:48:43 PM

To: TTAB EFiling

CC:

Subject: U.S. TRADEMARK APPLICATION NO. 85561168 - DEEP WEB INTELLIGENCE - 4335.14US01 - Request for Reconsideration Denied - Return to TTAB - Message 2 of 5

Attachment Information:

Count: 5

Files: RDDW2-03.jpg, RDDW2-04.jpg, RDDW2-05.jpg, RDDW2-06.jpg, RDDW2-07.jpg

time that the two genes can be connected by a chromosome using database SNP500Cancer. i.e., it turns out that the two genes are located in the same chromosome 10q11.2.

Overall, keyword queries over deep web data sources have specific features that are very distinct from keyword searches over relational databases. First, the number of deep web data sources in a domain can be substantially larger than the number of data tables in a relational database. For example, there are about 500 deep web data sources about SNP (Single Nucleotide Polymorphism) data, which is only a small branch of biology. Second, in many domains, it is common that a keyword query contains as many as 20 keywords, which rarely occurs in keyword queries over relational databases. Third, many relationships of interest can be derived only by querying across multiple data sources.

Our system integrates online biological deep web data sources and represents them as a multi-source inter-dependence hyper-graph. A novel bidirectional query planning algorithm generates multiple valid plans for answering keyword search queries. To address data redundancy, our system uses a domain ontology and a novel ranking strategy to select the most relevant set of data sources.

The rest of the paper is organized as follows. We describe our data model for keyword search in Section 2. In Section 3, we introduce our bidirectional query planning algorithm. The domain ontology and dependence graph ranking strategy are introduced in Section 4 and Section 5, respectively. In Section 6, we evaluate our system. We compare our work with related efforts in Section 7 and conclude in Section 8.

2. QUERY AND DATA SOURCE MODEL

In this section, we will introduce the query and data source model we consider in this paper.

2.1 Query Model

A query consists of $n, n > 1$, search terms t_1, t_2, \dots, t_n . The search terms are of two types, which are defined as follows.

Attribute Set AS: AS contains all attributes in the studied domain. An attribute is a part of the metadata (column name of the hidden databases) of a deep web data source. In other words, an attribute corresponds to an attribute of an entity in the ER diagram of the deep web data source's hidden databases. For example, suppose data source A has a

problem, the type of a keyword is determined by a domain ontology, which we will introduce in Section 4. Because the keywords involved in our searches are technical terms from a specific domain, it is relatively easy to decide the intention of each query. In generalizing our system in the future, we will incorporate the existing work [18, 26, 29] on this topic.

2.2 Data Model for a Single Deep Web Data Source

Each online deep web data source has a query interface and an output format. A user can construct a query by specifying some input attributes and constraints. Our model captures the above features of a deep web data source.

In our data source model, we view all the deep web data sources belonging to one sub-domain of biology (such as SNP) as a virtual relational table. In this virtual table, each data tuple is the query schema of a deep web data source. We call such a data tuple as a virtual data tuple. The virtual data tuples are connected by the inter-dependence relationship between deep web data sources.

Furthermore, if a single deep web data source has only one query interface (query schema), we model it as one virtual data tuple. If it has multiple query interfaces, we use multiple virtual data tuples to represent it. Each virtual data tuple is modeled as a record with three types of attributes, the input attributes, which are required in the query form, the output attributes, which are the attributes returned for the corresponding query, and the inherent constraints, which are the attribute conditions imposed on the data source by its designer. In addition, we separate the input attributes into two categories, the must-fill ones which have to be provided to get the query results and the optional ones which can be omitted and only provide extra constraint conditions to narrow down the search space. If the optional attributes are not provided in a query, we assume they will appear in the output attributes.

We denote a virtual data tuple formally as $R(MI, OI, O, C)$, where MI, OI, O and C correspond to the sets of must-fill attributes, optional attributes, output attributes and inherent constraints. For example, data source SeattleSNP has two query schemas, so it is modeled as two virtual data tuples as in Table 1. The first schema takes Gene Name as a must-fill input attribute, Up Base and Down Base as the optional attributes, and SNP Function and Frequency information as the outputs. It also has an inherent con-

Table 1: Data Model for SeattleSNP Data Source

Data	MI	OI	O	C
------	----	----	---	---

Data Source	MI	OI	U	C
Seattle Gene Name		Up Base Down Base	SNP Function Frequency	Organism= Human
Seattle SNPID Up Base		Down Base	Alleles Dis- equilibrium	Organism= Human

straint which means that all data in this data source are from human species. We notice that the model only contain high-level terms, not the real content of the database. For example, we only know that SeattleSNP can return SNP function information, but we do not know which SNP is included in this data source.

2.3 Data Model for Inter-Dependent Data Sources

Data sources are connected by the inter-dependence between them. For a certain query, a data source may need to be queried prior to another data source, so as to obtain the necessary input attributes of the second data source.

Consider a group of n deep web data sources,

$$R_1(MI_1, OI_1, C_1), \dots, R_n(MI_n, OI_n, C_n)$$

We assume all their attributes belong to a universal set of attributes. For two deep web sources R_i and R_j , we define three types of dependence relationships: Type 1) The query output of R_i can be applied to R_j 's must-fill input if $O_i \cap MI_j = \Phi$; Type 2) the query output of R_i can be applied to R_j 's optional input if $O_i \cap OI_j = \Phi$; Type 3) the optional input of R_i , which is also part of its output, can be applied to R_j 's must-fill input or optional input attribute, i.e., $O_i \cap (MI_j \cup OI_j) = \Phi$. The first type of relation shows that R_i has to be queried before R_j in order to obtain the necessary input attributes of R_j . The second type of relation shows that if R_i is queried before R_j , using the output from R_i , we can narrow down the searching scope of R_j or make the query on R_j more accurate. The third type of relation is the combination of the first two. These three types of dependencies play different roles when we are generating query answering plans. Figure 1 shows the inter-dependence among five deep web data sources for SNP data. Nodes are virtual data tuples, and three different types of arrows represent the three types of dependence relations above. We can see that dbSNP and Entrez protein form a hyper node for its descendant BLAST, which means that to be able to query BLAST, one needs to query both of dbSNP and Entrez Protein first.

Besides the dependence relationship between two data sources, R_i and R_j can also share common must-fill input attributes or output attributes. The first case implies that they share the same predecessor in the dependency graph. The second case implies that they can be the predecessors of the same descendant. We call this data redundancy.

3. BIDIRECTIONAL PLANNING ALGORITHM FOR KEYWORD SEARCH

We now consider the problem of keyword search over deep web data sources. For keyword search in traditional databases,

Figure 1: Dependence Relations between Five Data Sources

the set of data tuples in every relational table as nodes, and the foreign key relation between tuples as edges. Relational tables are also connected by foreign key relations forming a schema graph. Keyword query answering is done by using graph searching algorithms [14, 13, 17, 9, 22, 1, 2]

Algorithm 3.1: Bidirectional-Query-Planning(Q, N nodes)

```

Initialize IL,  $\forall u \in IL$ , add  $u$  to FQ, FE= $\Phi$ , FN= $\Phi$ 
Initialize BQ, BE= $\Phi$ , BN= $\Phi$ 
Create a pseudo-starting node PS
 $\forall u \in Nodes, \forall j \in Keywords$ 
if  $j \in O_u$ 
    distance $_{u,j} = 0$ 
    else distance $_{u,j} = \infty$ 
 $\forall u \in IL, \forall j \in Keywords$ 
if  $j \in O_u$ 
    decedents $_{u,j} = u$ 
    else decedents $_{u,j} = null$ 
 $\forall u \in Nodes, \forall j \in Keywords, sibling_{u,j} = null$ 
 $\forall u, v \in Nodes, BitMapGraph_{u,v} = 0$ 
if  $v \in IL$ 
    BitMapGraph $_{u,v} = 0$ 
    else BitMapGraph $_{u,v} = \infty$ 
ContinueSearch=true
while (FQ= $\Phi$  or BQ= $\Phi$ ) and continueSearch=true
    select the node with the highest priority score from FQ and BQ
    call it nextnode
    if nextnode  $\in IL$ 
        Explore(PS, nextnode)
    if Find Answer(PS)
        Plan Constructor()
        if doesn't need to generate more plans
            continueSearch=false
            else Refresh()
    if nextnode comes from FQ
        Forward-Explore(nextnode)
    else
        if nextnode  $\in IL$ 

```

web data sources. For keyword search in traditional databases, we have the direct access to the data. As a result, research in literature on keyword search with relational databases takes

```
if !isSingleNode()
  Backward-Explore(nextnode)
else Explore(PS.nextnode)
```

Algorithm 3.2: Forward-Explore(node)

```
foreach v in Neighbors(node)
  if v is LL
    Explore(PS,node)
  else
    U=Partners(node,v)
    if U is a single node
      Explore(U,v)
    if U is a hyper-node
      if for all n in U, n is FE or n is BE
        Explore(U,v)
      else put unexplored partners into BQ
    if v is FQ and v is FE
      add v to FQ
    if node is FQ
      decay the priority score of node
      add node to FN
```

Algorithm 3.3: Backward-Explore(node)

```
foreach U in Parents(node)
  if U is a single node
    Explore(U,node)
  if U is a hyper-node
    if for all n in U, n is FE or n is BE
      Explore(U,node)
    else put unexplored partners into BQ
  if node is FQ and node is FE
    add node to FQ
  if node is FQ
    decay the priority score of node
    add node to FN
  foreach n in U
    if n is BQ and n is BE
      add n to BQ
    if n is BQ
      decay the priority score of n
      add n to BN
```

Algorithm 3.4: Explore(U, v)

```
foreach keyword k
  if U is a single node
    if there is a shorter path from U to k via v
      nextnodeU,k = newdis, decedentU,k = v
      propagate the newdis to all reached ancestors of U
  if U is a hyper-node
```

Algorithm 3.5: FindAnswer(PS)

```
find=true
foreach keyword k
  if distanceU,k == 0
    find=false
return (find)
```

For deep web data sources, we do not know the data inside the hidden databases. So, keyword search cannot be directly performed on the data tuple granularity level. Instead, we need to address the keyword search problem at a higher-level (schema-level). Recall that in our data source model, we view all the deep web data sources belonging to one sub-domain (such as SNP) as a virtual relational table. In this virtual table, each data tuple is the query schema of a deep web data source, which we had earlier referred to as a virtual data tuple.

These virtual data tuples are connected by the inter-dependence relationship between deep web data sources. The number of deep web data sources of a sub-domain is substantially large in most cases, as we pointed out in Section 1, and many sources have multiple query interfaces corresponding to different query schemas. Thus, scale of the graph composed by virtual data tuples is comparable to the scale of the graph composed by real data tuples in traditional keyword search. By making this analogy, we want to adapt graph algorithms onto a higher granularity, in order to solve the keyword searching problem over deep web data sources.

Systems for keyword search on relational database, such as DBXplorer [1] and DISCOVER [14], model the query answering plan as a tree and have the direct access to the actual data. In a well defined relational database, there is not much data redundancy. As a result, the above systems do not consider data table ranking. The number of tuple sets for generating the candidate network used in DISCOVER system is proportional to the size of the power set of the keyword set. It is reasonable for keyword queries on relational database. In our scenario, we do not know the actual data inside the deep web data sources, and because some data sources can be queried in parallel (there is no dependency between them), therefore, a tree is not sufficient for us, instead, we need a forest structure to represent the query answering plan. Furthermore, we need to take the data redundancy into account to perform data source ranking. Finally, it is common to have keyword queries with more than

```
if U is a hyper-node
  compute an olddis which is the current shortest distance
  from any of the nodes in U to keyword k
  if there is a shorter path to k
    foreach n ∈ U
      nextnoden,k = newdis, decedentsn,k = v
    compute CA = CommonAncestor(U)
    foreach n ∈ U
      foreach anc ∈ Ancestor(n)
        if anc ∈ CA
          propagate newdis to anc
    foreach ca ∈ CA
      if ca is not an ancestor of another common
      ancestor in CA
        newdis = Max(BitMapGraphn,k
        + distancen,k), n ∈ U
        olddis = distancen,k
        if newdis < olddis
          update the distance
          propagate the distance to ca's ancestor
          decedentsn,k = U
          add sibling information to any node in U
```

10 or even 20 keywords. Due to these reasons, we need to propose a new algorithm to solve our problem.

One of the keyword searching algorithms is the bidirectional searching algorithm addressed in [17]. Since our data model is also a graph model, which is analogous with the model used in [17], we can address our problem by building on this bidirectional search algorithm. However, the characteristics of our dependence graph model give us many new challenges. First, unlike the foreign key relation in a relational database, our dependence relation is directed. This requires that our bidirectional algorithm is direction sensitive. Second, since the dependence relation is a multi-source relation, in our new algorithm, we not only need to keep track of the sequential order of node exploration, but also keep track of the parallel sibling relation between multi-source predecessors. Furthermore, we need to come up with a new edge exploration function which can deal with hyper-nodes. Finally, since in our problem, the query answering plan can be a forest with disconnected component, we need

Figure 2: Running Example of a Sample Query

multiple starting nodes to initiate the search on different component.

3.1 Algorithm Overview and Assumptions

Given a keyword query with n keywords and the dependence hyper-graph data model introduced in Section 2.3

It should be noted that our algorithm can be used on any domain. We assume that we have a ranking strategy for ranking each data source schema (node in graph) and a dependence relation (edge in graph). Furthermore, we assume that a domain ontology is built which can support the ranking strategy. The usability of the algorithm is independent

dependency hyper-graph data model introduced in Section 2.3, we need to first map the keywords onto the nodes in the dependency graph. Next, we want to traverse the graph using some algorithm to find multiple connected components which connect the mapped nodes to form a forest structure. Though we will discuss an example later, an example of the query answering plan forest can be seen in Figure 2. The traversing of the graph is done in a bidirectional manner. A forward exploration touches as many executable data sources as possible from a current data source. But, if one data source has many dependent data sources, it is likely to explore some unnecessary sources as well. If we are sure that some data sources should certainly be queried to answer the given query, we can view them as targets and perform a backward exploration.

To present the algorithm, the following concepts are defined.

Starting Node: From Section 2.1, we know that a query Q in our system must contain at least one entity name. The entity names are served as the triggering keywords which initiate the query. The answering of the keyword query must start from the data sources which can take the triggering keywords as their input. We call these data sources the starting nodes.

Data Source Necessity: Each data source has a set of output attributes. If an attribute can only be provided by a single data source, that data source should have a higher priority to be selected. Conversely, if the attribute can be provided by multiple data sources, a lower node score can be assigned to these data sources with respect to this attribute. Based on this idea, each term is associated with a Data Source Necessity value. Formally, for a term k , if R data sources can provide it as output, the data source necessity value for k is $DSN_k = \frac{1}{R}$. Then, the data source necessity value for a data source d is defined as follows $DSN_d = \text{Max}(DSN_k)$, $k \in O_d$.

ing strategy. The usability of the algorithm is independent of the ranking strategy and domain ontology proposed for this specific domain.

3.2 Algorithm Details

In this section, we first introduce the data structures we use in our algorithm, followed by the detail algorithm description. Algorithms 3.1, 3.2, 3.3, 3.4 and 3.5 show the pseudo-code.

3.2.1 Data Structures

Suppose there are N nodes in the dependence graph and K keywords in the query. Our bidirectional query planning algorithm uses the following data structures (some of these are based on [17]).

InitialNodeList IL: A list containing all the starting nodes of the search.

ForwardQueue FQ: A priority queue containing all nodes which are ready to be explored in the forward fashion.

ForwardExplored FE: A queue containing all nodes that have already been explored in the the forward manner.

ForwardNextRound FN: A priority queue containing all nodes which have already been explored in the forward fashion in the current round, and ready to be explored in the forward manner in the next round.

BackwardQueue BQ: A priority queue containing all nodes which are ready to be explored in the backward manner.

BackwardExplored BE: A queue containing all nodes that have already been explored in the the backward manner.

BackwardNextRound BN: A priority queue containing all nodes which have already been explored in the backward manner in the current round, and ready to be explored in the backward manner in the next round.

distance[N][K]: An array contains the shortest distance from any node to any keyword. The shortest distance is computed in terms of dependence graph edge score, which we will de-

fine later.

descendants[N][K]: An array maintains the next set of nodes that need to be visited in order to obtain the shortest distance from any node to any keyword.

sibling[N][K]: An array maintains a set of siblings (partners) needed to obtain the shortest distance from any node to any keyword. This data structure is designed for taking care of hyper-node predecessors.

BitMapGraph[N][N]: An array maintains the shortest distance for every pair of nodes in the graph.

3.2.2 Algorithm Initialization

currently, we cannot access v . Only when all predecessors are explored, we can call the Explore function to do edge exploration.

We perform edge exploration between U and v as follows. If U is not a hyper-node, the Explore function just updates the shortest distance information from U to any keyword via v and propagates the updated information to U 's ancestors if necessary. If U is a hyper-node, we use a different propagation strategy. We obtain all common ancestors of U to form a common ancestor set CA . A common ancestor ca is a node which is an ancestor of all nodes in U . For any node n in U , we propagate the distance information to ca via the edge (n, ca) . Then, we propagate the distance information from ca to all nodes in CA . The