Querying Deep Web Data Sources: A Structured Keyword Query Approach

Fan Wang [1], Gagan Agrawal [2]

Department of Computer Science and Engineering, The Ohio State University

[1]wangfa@cse.ohio-state.edu

[2]agrawal@cse.ohio-state.edu

***Abstract*— A popular trend in data dissemination involves online data sources that are hidden behind query forms, thus forming the *deep web*. Extracting information across multiple deep web data sources is challenging, but increasingly crucial in many areas. Keyword search, which is a popular information discovery method, has been studied extensively on the surface web and relational databases. Keyword-based queries can provide a powerful yet intuitive means for accessing information from the deep web as well. However, this involves many difficult challenges. For example, deep web data sources often contain redundant and/or incomplete data, there is often inter-dependence among data sources, and most data sources only support *non-predicate* keyword queries on their input interfaces. Thus, it is very hard to automatically execute complex queries.**

In this paper, we present a *structured keyword query* approach. We support keyword search over deep web data sources for complex queries, which can include *constraint predicates*, *aggregation functions*, *group-by operations*, and *nested sub-queries*. We have developed three query planning algorithms to generate query plans for different types of queries. Several optimization techniques are also used for optimizing the query plans and their execution.

I. Introduction

A popular trend in data dissemination involves online data sources that are hidden behind query forms, thus forming the *deep web*. Hundreds of large, complex, and in many cases, related and/or overlapping, deep web data sources have become available. The common way of accessing data in deep web data sources is through standardized input interfaces. These interfaces, on one hand, provide a very simple interface, and do not require that the users know any specific query language and/or the schema(s) of the underlying databases. On the other hand, these interfaces significantly constrain the types of queries that could be automatically executed. For example, in the biological domain, most data sources only have one text box in their input interfaces. Such text boxes only accept *simple text*, i.e., the users cannot specify any constraints, aggregation functions, or nested queries. Furthermore, many useful queries involve accessing information from multiple data sources, and currently, there is no available support for this class of queries.

Thus, it is clearly desirable to support more complex queries across multiple deep web data sources. In recent year, a number of efforts have been attempting to build deep web systems that can integrate both the query interface and the query results (structured data) of the deep web-sites within a specific domain [1], [2], [3], [4]. Some other systems mainly deal with *optimizing* the execution of cross-source queries on the web [5], [6], but do not consider query planning starting from intuitive queries with high expressive power. The systems proposed in [7], [8] have query planning algorithms, but they either ignore complex queries with value constraints or aggregation [8] or can only support constraint predicates that match the filters supported by data sources locally [7]. The problem we are considering in this paper also has some similarity to the work on mediators systems like SIMS [9], Information Manifold [10], TSIMMIS [11], and MedMaker [12]. However, in considering the deep web, the data source model and the cost metric are very distinct, and as a result, the query planning problem formulation and algorithms are different.

This paper focuses on addressing the problem of supporting intuitive queries with high expressive power on a set of integrated deep web data sources within a domain. Our premise is that keyword queries can provide an intuitive yet powerful means for expressing complex queries over a set of deep web data sources. Keyword search has been a very popular information discovery method over the surface web. In recent years, it has been applied on relational and graph datasets as well [13], [14], [15], [16]. Recently, a system SOAK [17] has been developed, which is able to handle keyword queries with aggregations on relational databases. Our work is driven by gaining popularity of this query mechanism, but considers two additional goals. Our first goal is to support keyword queries, with aggregation, constraints, and the possibility of nested queries. Second, unlike any of the existing work on keyword search, we are considering such queries over deep web data sources. Considering queries with more expressive power and over a different set of data sources involves a number of challenges. We consider two motivating examples to show both the desirability and difficulty of supporting such queries.

Motivating Example 1: Suppose we are interested in Single Nucleotide Polymorphisms (SNPs), which are particularly promising for explaining the genetic contribution to complex diseases [18]. Biologists have identified that the gene X and the protein Y are contributors of a disease. Now, they want to examine the SNPs located in the genes that share the same functions as either `X` or `Y`. Particularly, for all the SNPs located in each such gene functions similar to either `X` or `Y`, and those that have a heterozygosity value (which is a measure

```
select Max(s.Asian_Allele_Frequency)
from SNP s, GENE g
where s.Heterozygosity > 0.01 AND s.gene=g.gene
        AND g.function IN
        (select g1.function                 (select g2.function
         from GENE g1          OR            from GENE g2
         where g1.name=X)                    where g2.proteinname=Y)
group by (g.name)
```

Fig. 1. SQL Query for the Motivating Example

of the genetic variation in a population) greater than 0.01, biologists wants to know the *maximal* allele frequency in the Asian population.

To understand this example, suppose we have relational tables SNP and $GENE$ containing all SNPs and Gene (or Protein) related information. If this is the case, we can express the above query in an SQL-like fashion, which we show in Figure 1.

In practice, however, this information is available across multiple different deep web data sources. Furthermore, this query requires an aggregation function, a group by operation, value constraints and even nested sub-queries. Specifically, we need to take the following steps, which are also shown graphically through a query plan in Figure 2.

1. The *first sub-query*: find the genes that have the same function as the gene X (query-plan part 1, sub-figure (a), Figure 2).

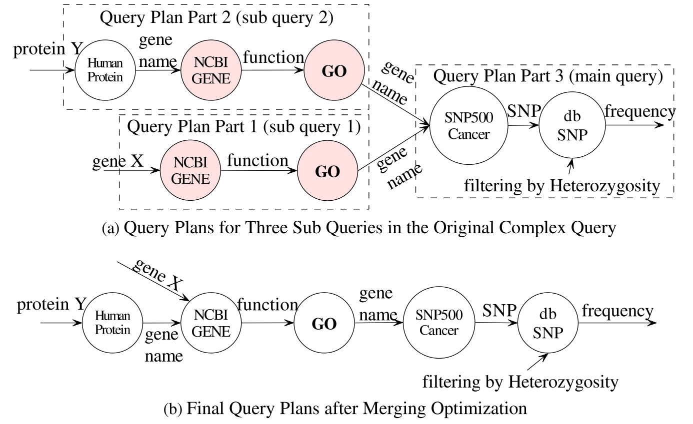
2. The *second sub-query*: find the genes that have the same function as the protein Y (query-plan part 2, sub-figure (a), Figure 2).

3. The *main query*: for each gene obtained from the steps 1 and 2, find all the SNPs, filter out the SNPs with Heterozygosity value smaller than 0.01, and find the maximal allele frequency in Asian population (query plan part 3, sub-figure (a), Figure 2).

Specifically, `NCBI Gene` and `Human Protein` data source takes gene `X` and protein `Y` as input, respectively. The names of the genes with the same function as `X` and `Y` will be obtained from the `GO` data source in the query plan parts 1 and 2. Taking the genes obtained from `GO`, and using the `SNP500Cancer` data source, we can find the human SNPs located on these genes. Finally, using the `dbSNP` data source, we can find the SNP frequency and heterozygosity information. Then, the maximal allele Asian frequency of the SNPs in each gene can be found by an aggregation computation, while filtering out all SNPs that have heterozygosity value smaller than 0.01.

Motivating Example 2: A travel agent in a mid-west city X wants to advertise new travel destinations on the west coast. He is interested in finding the answers to the following query *"find all west cost cities that have an average airfare (from city X on particular dates) that is lower than the average airfare to Los Angeles"*. To answer this query, we need to perform two aggregations. Furthermore, X to Los Angeles is a sub-query that provides a condition value for the main query.

From the above two examples, we observe that to answer complex queries on deep web data sources, users face the



(a) Query Plans for Three Sub Queries in the Original Complex Query

(b) Final Query Plans after Merging Optimization

Fig. 2. Motivating Example: (a) Query Plans for Three Sub Queries, (b) Final Merged Query Plan

following difficult challenges. First, they need to be familiar with domain data sources and know which data sources are relevant to the query. On similar lines, they may even need to know the inter-dependence between data sources (i.e., when the output of one data source may be the necessary input to another one), so as to know the order of querying data sources. Second, since some data sources may have redundant data [19], users need to manually rank similar data sources. Third, they need to manually keep track of all the results from multiple data sources. Finally, since most deep web data sources today only accept simple text queries, they need to filter out data tuples violating constraints and perform group-by and aggregation functions. Clearly, this requires a great deal of knowledge, involves a large amount of manual work, and thus can be a tedious and error-prone process. It is desirable that a system can automate this process, at least over a set of deep web data sources within a domain or sub-domain.

In this paper, we have developed a *structured keyword query* approach for querying a set of integrated deep web data sources. In our approach, users can write complex queries by specifying keywords, with necessary predicates and nesting structures. Currently, our approach supports the following SQL-like query features: *value constraints*, *group by operations*, *aggregation functions*, and *nested queries*. For example, the query corresponding to the first motivating example could be expressed as
"*MAX(Asian_Allele_Frequency),Heterozygosity>0.01, Gene{Ontology_Function,X}OR{Ontology_Function,Y}*".

Our approach for supporting these queries is as follows. For each nested sub-query in the user query, a query plan is generated. We have query planning algorithms for the three types of queries we consider. Finally, the query plans for all sub-queries are *combined* and/or *merged* together to form an entire query plan. For example, the sub-figure (b) in Figure 2 shows the final plan after merging similar data sources `NCBI Gene` and `GO` from two nested query plans. Next, several optimization techniques that we have developed are applied to speedup the query plan execution.

While our implementation and the examples used in this paper are based on the structured keyword queries, our overall contributions are broader. Our query planning algorithms and optimizations would still be applicable if a different format was used. For example, we can easily support the use of SQL for the same set of queries.

Overall, the main contributions of this paper can be summarized as follows:

1. To the best of our knowledge, our work is the first one to handle complex queries, including user-defined constraints, group-by operations, aggregation functions, and nested queries across multiple deep web data sources.
2. We have developed query planning algorithms to generate query plans for different types of queries.
3. Several optimization techniques have been developed for query planning and query execution.

The rest of the paper is organized as follows. In Section II, we introduce the structured keyword query model used in our approach. The data model used in our query planning algorithm is described in Section III. In Section IV, three query planning algorithms are presented. Several optimization techniques for query planning and query execution are described in Sections V and VI, respectively. Our techniques are evaluated in Section VIII. We compare our work with related efforts in Section IX and conclude in Section X.

II. Structured Keyword Query Model

In this section, we first describe the basic *Structured Keyword Query* (SKQ) model, and then we extend the basic SKQ model to the *full* SKQ model.

A. Basic Structured Keyword Query Model

The basic idea that has initiated this work is that a simple and restricted class of *select-from-where* queries in SQL can also be represented as keyword queries. For this, we define a *basic SKQ* as $Q = \{a_1, \ldots, a_k, e_1, \ldots, e_m\}$. Here, each a_i is an *attribute keyword*, which corresponds to attributes in the selection clause of the corresponding SQL query. Also, each e_i is an *entity keyword*, which corresponds to entities or rows returned in the where clause of an SQL query. The intent of a *basic SKQ query* is to obtain the information on entities $e_1, \ldots, e_m$, specifically, the values of the entities' attributes $a_1, \ldots, a_k$.

For example, the query "find the allele frequency of SNPs located in Gene `ERCC6`" can be written as a basic SKQ "*Allele_Frequency, ERCC6*", where *Allele_Frequency* is an attribute keyword and *ERCC6* is an entity keyword.

B. Improving Expressive Power

We extend the expressive power of basic SKQ by incorporating aggregation functions and comparison predicates.

Aggregation Functions: Five aggregation functions are currently supported, which are AVE(a), SUM(a), COUNT(a), MAX(a), and MIN(a). a is considered the *aggregation attribute*.

For example, the query "find the minimal Heterozygosity among all SNPs located in gene `ERCC6`" can be expressed as "*MIN(Heterozygosity), ERCC6*".

Comparison Predicates: Standard comparison predicates, $=$, $>$, $\geq$, $<$, and $\leq$ are supported. An attribute associated with a comparison predicate is considered to be a *condition attribute*.

As an example, the query "find the alleles of the SNPs with a heterozygosity value greater than 0.01 and located in gene `ERCC6`" can be expressed as "*Alleles, Heterozygosity* $>$ *0.01, ERCC6*", with *Heterozygosity* being the *condition attribute*.

C. Nested Sub-Queries and Full Model

The basic SKQ is extended to a full SKQ by applying any comparison predicate or an aggregation function to an *attribute keyword* a_i in Q and/or by adding nested entity or condition queries. The former were described in the previous subsection. We now describe the type of nested sub-queries we allow.

Nested Sub-Queries: We support two types of nested sub-queries, which are the *condition nested* and *entity nested* queries. A nested query is enclosed by $\{$ and $\}$. If a nested query is a condition nested query, its output is used as a constraint value in a comparison predicate in the parent query. If a nested query is an entity nested query, its output is used to specify an entity keyword in the parent query.

For example, the query "find the alleles of the SNPs located in `ERCC6` and with a heterozygosity value smaller than the largest heterozygosity value among all SNPs in gene `APOE`" can be expressed as

"*Alleles, Heterozygosity*$<\{$*MAX(Heterozygosity), APOE*$\}$*, ERCC6*"

Here "*MAX(Heterozygosity), APOE*" is a condition nested query that specifies the value for the condition attribute *Heterozygosity* in the outer (parent) query.

As another example of nested query predicates, the sub-query "*Ontology_Function, X*" in the motivating example listed in the previous section is an entity nested query. It finds all the genes having the same function as `X` for the outer query.

Nested Query Relation Predicates: We supports two types of nested query relations, OR and AND. The query in the example in Section I uses the OR predicate.

Example of Full Model: To consider a more detailed example, we can revisit the first motivating example in Section I. The keyword query here is extended from the basic SKQ ("*Asian_Allele_Frequency, Heterozygosity, Gene*")

by applying the MAX aggregation predicate to the attribute keyword *Asian_Allele_Frequency*, applying the $>$ comparison predicate to *Heterozygosity*, and adding two nested entity queries *"Ontolgo_Function, X"* and *"Ontology_Function, Y"* to specify the entity keyword *Gene*.

Tree Representation: We can express an SKQ with a tree structure, which is denoted as the *SKQ tree*. If a SKQ Q does not contain any nested queries, we consider Q to be a *simple query*. In this case, the SKQ tree is a tree with a single node. If Q contains nested sub-queries, according to the nested query predicates in Q, we partition Q into multiple sub queries $subQ_1, \ldots, subQ_n$, and a main query $mainQ$. Then, the node for $mainQ$ is the root node for the tree. We build a node n_i

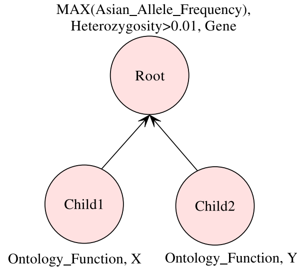


Fig. 3. SKQ Tree for the Query in the Motivating Example (Section 1)

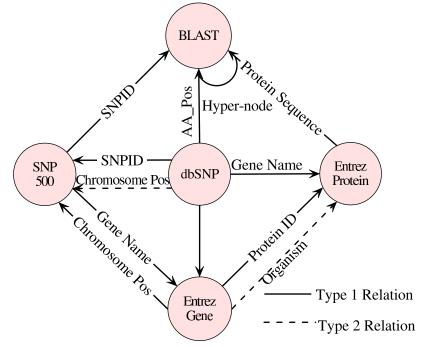


Fig. 4. Dependence Relations between Five Data Sources

TABLE I
DATA MODEL FOR `Seattle` DATA SOURCE

Data Source	MI	OI	O	C
Seattle1	Gene Name	Up_Base Down_Base	SNP_Function Frequency	Organism= Human
Seattle2	SNPID	Up_Base Down_Base	Alleles Dise-quilibrium	Organism= Human

for each sub-query $subQ_i$, and link n_i to the root as its child. Finally, we build the rest of the SKQ tree for each of the child node n_i recursively. For example, the SKQ tree for the query in the motivating example in Section I is shown in Figure 3.

III. DATA SOURCE MODEL

Our system is designed to execute the queries described in the previous section over a set of deep web data sources from a domain or sub-domain. To enable the planning and execution of queries, we need to know the schemas for each of the data sources and the inter-dependence between them. Formally, this is captured through a *data source model*, which we will describe here.

A. Data Source Model for a Single Deep Web Data Source

A deep web data source may have multiple input query interfaces and each input interface corresponds to a unique output schema. In our model, we view a data source as a *virtual relational table* with a set of data tuples. Each data tuple is formally defined as $R(MI, OI, O, C)$. MI refers to the must-fill input attributes which have to be provided to get the query results. OI refers to the optional input attributes which can be omitted and only provide extra constraint conditions to narrow down the search space. O refers to the output attributes returned for the corresponding input. C refers to the attribute conditions imposed on the data source by its designer. For example, the model of data source `Seattle` is shown in Table I. `Seattle` has two input interfaces take `Gene Name` and `SNPID` as input respectively, and `Seattle` only contains data from human species as shown in the C column of the table.

B. Data Model for Inter-dependent Data Sources

Data sources are connected by the inter-dependence between them and form a hyper-graph dependency model. For two data sources R_1 and R_2, we define two types of dependence relationships: Type 1) R_1 can provide must-fill input for R_2, i.e., $O_1 \cap MI_2 \neq \Phi$; and Type 2) R_1 can provide optional input for R_2, i.e., $O_1 \cap OI_2 \neq \Phi$. The first type of relation shows that R_1 has to be queried before R_2 in order to obtain the necessary input attributes of R_2. The second type of relation shows that if R_1 is queried before R_2, using the output from R_1, we can obtain optional attribute of R_2 which can be used to narrow down the search scope of R_2 or make the query on R_2 more accurate.

Some data source dependencies are multi-source, i.e. the input of a data source depends on the output from multiple data sources, as a result, our dependence graph model is a *hyper-graph*. In Figure 4, we can see that `dbSNP` and `Entrez protein` form a *hyper-parent* node for `BLAST`, which means that to be able to query `BLAST`, one needs to query both `dbSNP` and `Entrez Protein` first.

Somewhat similar to our work, Davulcu *et al* [20] proposed a *navigation map* to capture the link structure between web pages generated by deep web data sources. The main difference of our dependence model is that we model the *inter data-source* dependencies, but they model the *intra-data source* web page dependencies.

IV. QUERY PLANNING PROBLEM: FORMULATION AND ALGORITHMS

In this section, we first give an overview of the query planning problem for the structured keyword queries we are considering. Then, we formally state query planning problem for the three types of queries, and present the query planning algorithms for each of these.

A. Query Planing Problem Overview

We have a structured keyword query (SKQ) Q and a data source dependency hyper-graph DG as described in Section III-B. We want to find a query plan for Q. Because there may be multiple nested queries in Q, we propose a *divide-and-conquer* approach. We first build a tree for Q as we had shown earlier. Next, we perform a depth-first traversal on this tree, and generate a query plan for each node of the tree, which represents a *simple query*. The final query plan for Q is obtained by combining and merging all the query plans thus generated.

The last of these steps is described in the next section. This section focuses on the problem of generating query plans for different types of simple queries.

In our query planning problem, we consider three types of simple queries. This includes two types of simple queries that require special treatment, and a third *default* or *ordinary* case. The first type that has a different requirement for query planning is an *aggregation query*, which is a query with an aggregation function. The second type that requires a separate procedure is a nested entity sub-query. The last type is the *ordinary query*, which is a query that is neither an aggregation query nor a nested entity sub-query. Ordinary SKQ query is also a basic query, with the possibility of condition predicates. It can also be a condition nested sub-query, but cannot be a nested entity sub-query.

For example, consider parts of the example query in Section I. "*MAX(Asian_Allele_Frequency), Heterozygosity>0.01, Gene*" is an aggregation query, and " *Ontology_Function, X*" and "*Ontology_Function, Y*" are nested entity sub-queries.

Each type of simple query has specific requirements for a valid query plan. Thus, they need to be treated separately. In the rest of this section, we will present algorithms for each of these three types of queries. The overall query planning algorithm for an entire query tree is based on invoking these algorithms, and a *combine* function which will be presented in the next Section. This overall procedure is shown as Algorithm IV.1.

Algorithm IV.1: SKQ-Query-Planning($SKQTreeT$)

if $T.child = null$ */*T is a simple query*/*
  if T.query is an ordinary query
    $Final - Plan = Bidirectional(T.query)$
  if T.query is an aggregation query
    $Final - Plan = Center - Spread(T.query)$
  if T.query is a nested entity query
    $Final - Plan = Nested - Entity(T.query)$
else */*T has nested queries*/*
  foreach $T_i \in T.children$
    $Plan_i = SKQ - Query - Planning(T_i)$
$Final - Plan = CombinePlan(Plan_1, .., Plan_n)$

B. Query Planning for Simple Ordinary Query: Formulation and Algorithm

According to the definition of a simple ordinary query, it has the same format as a basic query introduced in Section II-A, except that a simple ordinary query could contain attribute value constraint, such as "*Heterozygosity>0.01*". A value constraint VC can be handled in two ways. First, if the VC matches any data source constraints specified by the C column in the data source model, the VC can be used as a condition for data source ranking. Alternatively, i.e., if VC does not match any data source constraint, it can be used as a *data tuple filter* during the plan execution.

For the rest of this discussion, we do not focus on the possibility of having to filter data tuples. Thus, given a simple query Q with n keywords and a data source dependency graph DG, the query planning problem is as follows. We want to find a *minimal* subgraph, or a subgraph with the fewest nodes, $SubG$ from G, such that the set of all output attributes from the nodes in $SubG$ covers all attribute keywords in Q, and the set of all input attributes of the nodes without incoming edges (the starting nodes) in $SubG$ covers all entity keywords in Q.

The reason why such a subgraph $SubG$ forms a valid query plan for Q is as follows. For an attribute keyword k_a in Q, we want to obtain its value from certain data sources, so we at least one node in $SubG$ that outputs (or covers) k_a. For an entity keyword k_e, k_e helps *initiate answering* of Q. Thus, we need the input of the *starting nodes* in $SubG$ to cover k_e.

For example, consider the entity keyword *ERCC6* in the query "*SNPID, Gene_Function, ERCC6*", where the intent is to find the SNPs and gene functions of the gene `ERCC6`. Using a domain dictionary or ontology, we can know that *ERCC6* is a gene name. As a result, we want a data source node whose set of input attributes includes the attribute `Gene Name`.

Further, the reason we choose the sub-graph with the smallest number of nodes is as follows. In the deep web, querying each additional data source implies another query over the network. Thus, using the fewest data sources is the key goal in query planning.

Unlike a query plan for keyword query in relational databases[13], [14], a query plan in our case is a *subgraph*, but not a *tree*. The reason is that, for queries defined as in Section II, our goal is not to connect all attribute keywords as in relational database keyword search. Instead, our goal is to connect all attribute keywords with entity keywords through data sources so as to obtain the values of attribute keywords. As a result, a query plan in our case can even be a graph with disconnected components.

Formulation for Ordinary Query: Given an ordinary SKQ Q that contains n keywords, the query plan of Q is a minimal (smallest size) subgraph, $SubG$, as formulated above. We have established the following result.

Lemma IV.1. *The query planning problem for ordinary SKQ is NP-hard.*

Proof. The standard *set cover* NP-hard problem can be reduced to the *ordinary SKQ query planning* problem. We will describe how to construct a graph (query plan) from any instance of the set cover problem. Further, we will show how, in this graph, any optimal solution for the set cover problem corresponds to an optimal solution to the ordinary SKQ query planning problem.

Obviously, given a subgraph (query plan) of a graph G (dependency graph), we can check whether this subgraph covers a set in polynomial time. This shows that our problem belongs to NP. Next we will show it is also NP-hard.

Let $C = \{C_1, C_2, \ldots, C_m\}$, be an instance of the set cover problem. Recall that a solution to the set cover problem is a minimum subset $C^{'} \subseteq C$, such that each element of $S = \bigcup_{i=1}^{m} C_i$ is contained in C'. Denoting the elements of S by $s_1, s_2, \ldots, s_n$, the graph $G = (V, E)$ is constructed as follows.

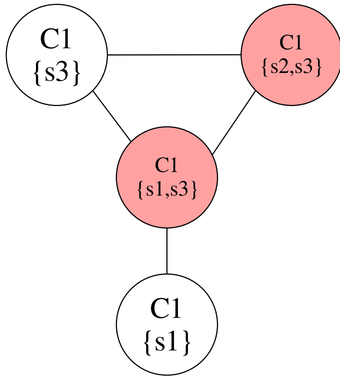


Fig. 5. Graph Example for the Proof of Lemma IV.1

The node set V of G is given by

$$V = \{C_1, \ldots, C_m\}$$

Here, C_i is a set in the set cover problem. Let each C_j node covers the elements which are covered by the set C_j in the set cover problem. The given set T is defined as $\{s_1, \ldots, s_n\}$.

The edges in G is constructed as follows. For each pair of nodes C_i and C_j in G, if the element sets covered by them have common element(s), the two nodes C_i and C_j are connected by an edge. Obviously, the graph G can be constructed in polynomial time. Figure 5 shows an example for the set cover instance:

$$\{\{s_1, s_3\}, \{s_1\}, \{s_3\}, \{s_2, s_3\}\}$$

Observe that any subgraph of G that covers all elements in $T = \{s_1, \ldots, s_n\}$, must contain some C_j nodes. A critical observation is that in order to make the size of the subgraph minimal, we must use the minimal number of C_j nodes. But, now, this is equivalent to minimizing the set cover represented by C_j. This shows that any set of C_j nodes that are contained in the minimal connected subgraph of G, such that it covers all elements in T, is a minimal set cover also, and vice-versa. In Figure 5, the shaded subgraph shows an instance of the minimal subgraph covering T, which also forms a minimal set cover.

□

Bidirectional Query Planning Algorithm: We use a heuristic bidirectional planning algorithm to find query plans for ordinary SKQ, which is based on the original bidirectional algorithm for keyword search on relational databases by Kacholia *et al.* [21].

1) Algorithm Overview:: We consider the data source nodes that cover *attribute keywords* as the *target nodes* and the data source nodes that cover *entity keywords* as the *starting nodes*. A query plan ultimately connects a subset of target nodes with a subset of starting nodes, such that that all keywords are covered. We explore the query plan in a bidirectional manner. We perform backward exploration from the target nodes to connect them with starting nodes. To accelerate this process, we also do forward exploration from the starting nodes. In this way, the bidirectional exploration can meet *mid-way*.

Benefit Function and Heuristics: We have a benefit function defined for data sources to rank similar data sources and decide which data sources are more relevant to the query. Intuitively, the benefit function ranks a data source higher if the data source has the following properties: 1) it can cover more search terms, 2) it provides data of *higher quality*, as determined by an ontology, and/or 3) its underlying constraints satisfy the value constraints VC in the query. In order to find the subgraph covering all search terms with the minimal size, the heuristic we use is the following. The edges that allow a shorter distance to that nodes that cover keywords are preferred. The benefit function and the heuristic outlined above are also used for the other two planning algorithms, which will be presented in in Sections IV-C and IV-D, respectively.

2) Bidirectional Exploration:: Initially we add all *starting nodes* to a forward exploration queue, and all *target nodes* to a backward exploration queue. Initially, if a node covers a search term, the distance of reaching this search term from the node is 0, otherwise the distance is infinity. For each node, we use the node ranking function to compute a node score to indicate the relevance of this data source w.r.t. the query Q. At each iteration, the planning algorithm selects the highest ranked nodes, CN, from the two queues. If CN belongs to the forward queue, all out-going neighbors of CN will be explored using *forward exploration*. If CN belongs to the backward queue, all in-coming parents of CN will be explored using *backward exploration*. During the backward or forward exploration, since new nodes are explored, the distance from a data source to search terms could be updated. This is done using *edge exploration*. Finally, when every search term can be reached from a starting node with finite distance, a query plan is found and this query plan is a graph with disconnected components, each of which is rooted at the corresponding starting node. The details of forward exploration, backward exploration and edge exploration are introduced as follows.

Forward and Backward Exploration: Forward exploration explores the edge e between CN (predecessor) and one of CN's descendants. Backward exploration explores the edge e between CN (descendant) and one of CN's predecessor. Regardless of the direction of the exploration, for an edge e, we denote the predecessor node as u and the descendant node as v. Since our dependency graph is a hyper-graph, when e is explored, we need to consider two cases. In the first case, the predecessor u is a single node. We just directly perform an *edge exploration* on e. For example, in Figure 4, suppose we are now on node `dbSNP`, and we want to do a forward exploration to `Entrez Gene`. Since `dbSNP` is the single predecessor of `Entrez Gene`, the exploration can be done directly. All search terms covered by `Entrez Gene` now can be reached from `dbSNP` via `Entrez Gene`, so that the distance from `dbSNP` to the search terms originally covered by `Entrez Gene` should be updated using the *edge exploration* function. In the second case, u is involved in a hyper-node with respect to v. In this case, all nodes in the

hyper-node should be explored previously so that the hyper-edge between the hyper-node and v can be explored. This is because the accessibility of the dependent node v depends on the accessibility of all its predecessors in the hyper-node. As a result, if any node in the hyper-node is not accessible currently, we cannot access v. In order to explore edge e, we need all the unexplored nodes in the hyper-node to be explored. From this point of view, these unexplored nodes become *target nodes* and, therefore, are added to the backward queue. For example, in Figure 4, suppose we are on `dbSNP`, and we want to do a forward exploration to `BLAST`. Since `dbSNP` and `Entrez Protein` are joint hyper-predecessors for `BLAST`, we need both of `dbSNP` and `Entrez Protein` been explored so as to explore `BLAST`. If `Entrez Protein` has not been explored, we can consider `Entrez Protein` as a new *target node* and add it to the backward queue.

The backward exploration is executed in similar way as the forward exploration. After a backward exploration, suppose from v to u, we add node v to the forward queue to explore the frontier of v.

Edge Exploration: The goal of edge exploration is to update the distance information, i.e. the distance from starting nodes to search terms, whenever a new node is explored. The distance of an edge e is computed by our *edge ranking* function, which will be introduced in Section VII-E. An edge with a shorter distance is preferred and the shorter distance is propagated to the starting nodes. Suppose the two end nodes of an edge e are u (predecessor) and v (descendant). The edge exploration is performed in two steps. The first step is the *local distance update* on the predecessor node u, and the second step is *distance propagation* to u's ancestors. If u is a single node, we first locally update the shortest distance from u to any search term could be reached via v. Then, we propagate the updated distance to u's ancestors. This is the standard edge relaxation in Dijkstra algorithm.

If u is a hyper-node, we use a different strategy. We first locally update the shortest distance on the hyper-node u. For the propagation phase, we differentiate u's ancestors into two types. The first type of ancestors are the *Shared Ancestors* or SA, which are the common ancestors shared by more than one node in the hyper-node u. The second type of ancestors are the *Unshared Ancestors* or UA, which are the ancestors for a single node in u only. An important feature for a shared ancestor, sa, is that the shortest distance from sa to a search term could be reached via u depends on the *longest* path among all the paths connecting sa to each node u_i in u. As a result, we perform distance propagation on *unshared ancestors* as normal. For *shared ancestors*, we first compute a batch of shortest distances from sa to u using every node u_i in u. Then, we propagate the *longest one* to sa. Taking Figure 4 as an example, suppose all edges have distance of 1 except for the edge between `BLAST` and `dbSNP` having a distance of 2. The search term `ORTH BLAST` is covered by `BLAST` and we want to update the distance information of `ORTH BLAST` to `Entrez Gene`. Because `dbSNP` and `Entrez Protein` are joint predecessors for `BLAST`, although the distance from `BLAST` to `Entrez Gene` via `Entrez Protein` is 2, the final distance should be 3 which is through `dbSNP`.

Algorithm Termination: When every keyword can be reached from a starting node with finite distance, a query plan is found.

C. Query Planning for Simple Aggregation Query: Formulation and Algorithm

In an aggregation query, we define an attribute to be an *grouping attribute* if it is neither an aggregation attribute (i.e. an attribute associated with an aggregation function) nor a condition attribute (i.e. an attribute associated with a comparison predicate). For our current implementation, we assume that we have only one grouping attribute for an aggregation query. We further define a data source whose output covers any aggregation attribute to be an *aggregation data source*. Similarly, a data source whose output covers the grouping attribute is considered the *grouping data source*. In order to perform aggregation functions on the groups formed by grouping attribute, we must know the mapping between the grouping attribute and the aggregation attributes. For example, in the query "*MAX(Heterozygosity), Gene_Name, Chromosome10*", we need to group all SNP heterozygosity values according to gene name and apply the MAX function. This requires that we know which SNP maps to which gene. In other words, if SNP and gene information are covered by two data sources, these two sources must be *connected*, so that we can discover the mapping between SNPs and genes.

A correct query plan for this query is shown as sub-figure (a) in Figure 6. We can see that the data source A, which covers *Gene_Name*, is connected with the data source C, which covers *Heterozygosity*. Sub-figure(b) in Figure 6 shows another query plan, in which the aggregation data source (F) is not connected with the grouping data source (E). If this query plan is to be used, we can obtain the genes data and SNP data located in the chromosome 10 from E and F, respectively, but we cannot know which SNP is located in which gene. As a result, we cannot perform the grouping operation on *Gene_Name*.

Thus, we can state that for a simple aggregation query, a valid query plan must meet the *Node Connection Property*, which requires that the data sources that cover the aggregation attribute(s) must be directly or indirectly connected with the data source that covers the grouping attribute in the query plan. In the query plan of an aggregation query, if the aggregation data sources are not connected (directly or indirectly) with the grouping data source, the query plan is considered having an error. The plan in sub-figure (b) has such an error.

Formulation for Aggregation Query: Given an aggregation query Q that has an aggregation keyword set $AggK$ and a grouping keyword $groupk$, we want to find a minimal sub-graph, $SubG$, as formulated in Section IV-B, which also satisfies the node connection property. We have established the following result.

Lemma IV.2. *The query planning problem for a simple aggregation SKQ is NP-hard.*

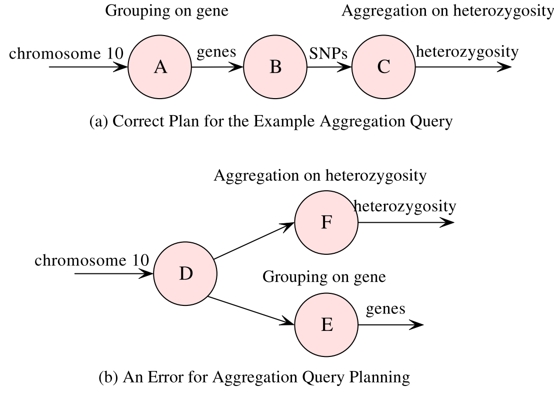


Fig. 6. Query Plan Example for Aggregation Query: (a) Correct Plan for the Example Aggregation Query, (b) Plan with an *Error*

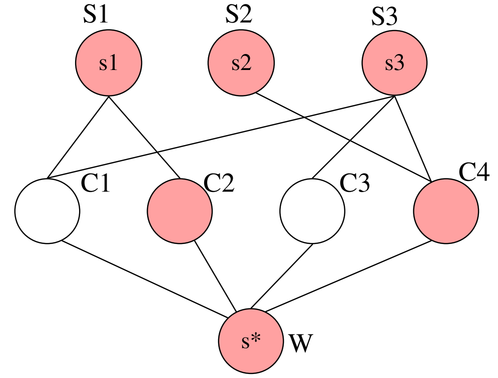


Fig. 7. Graph Example for the Proof of Lemma IV.2

Proof. The standard *set cover* NP-hard problem can be reduced to the *aggregation SKQ query planning* problem.

Using the same settings as in the proof of Lemma IV.1, let $C = \{C_1, C_2, \ldots, C_m\}$, be an instance of the set cover problem. Denoting the elements of S by $s_1, s_2, \ldots, s_n$, the graph $G = (V, E)$ is constructed as follows. The node set V of G is given by

$$V = \{s_1, \ldots, s_n, C_1, \ldots, C_m, W\}$$

Here, C_i is a set in the set cover problem, and W is a new node added to G. Let each s_i node in G cover the element s_i. Among the n elements, we suppose the element s_g is the grouping attribute and all rest elements are aggregation attributes. Thus, the node corresponding to element s_g is the grouping node, and all other s_i node(s) are aggregation node(s). Let each C_j node covers nothing and the node W covers a special attribute s^*. The given set T is defined as $\{s_1, \ldots, s_n, s^*\}$.

G contains two sets of edges. First, for each of the C_i nodes, there is a directed edge connecting the C_i to the W node. Second, each s_i node is connected with any node C_j, for which $s_i \in C_j$. Obviously, the graph G can be constructed in polynomial time. Figure 7 shows an example for the set cover instance:

$$\{\{s_1, s_3\}, \{s_1\}, \{s_3\}, \{s_2, s_3\}\}$$

Observe that any subgraph of G that covers all elements in $T = \{s_1, \ldots, s_n, s^*\}$ and satisfies the node connection property, i.e., the s_g node (grouping node) must be connected with all other s_i nodes (aggregation nodes), must contain all s_i nodes as well as the W node. This is because the s nodes need to be connected through the W node. In the example in Figure 7, we assume the node s_1 and s_2 are aggregation nodes, and the node s_3 is the grouping node. Given the nature of the graph, this will only be possible through the inclusion of some C_j nodes. A critical observation is that in order to make the size of the subgraph minimal, we must use the minimal number of C_j nodes. But, now, this is equivalent to minimizing the set cover represented by C_j. This shows that any set of C_j nodes that are contained in the minimal connected subgraph of G, such that it covers all elements in T, is a minimal set cover also, and vice-versa. In Figure 7, the shaded subgraph shows an instance of the minimal subgraph covering T satisfying the node connection property. At the same time, the shaded C_j nodes form a minimal set cover.

□

Center-spread Query Planning Algorithm: To ensure the *node connection property* in a query plan, we start building a query plan from the aggregation data sources and gradually add other data sources to the query plan if they are connected to the aggregation data sources via some path. This procedure continues until all keywords are covered by data sources in the plan. We call this algorithm the *center-spread query planning algorithm*. We describe the center-spread algorithm below, and show the pseudo-code in Algorithm IV.2.

Algorithm Initialization: Same as the bidirectional algorithm, we have *starting nodes* and *target nodes* initialized in this algorithm (line 1). At the beginning, we add all the aggregation data sources into the query plan. We also have a *center neighbor queue* CNQ containing all the data sources that are *directly connected* with at least one data source in the query plan (initially the CNQ contains the direct neighbors of aggregation data sources). Here, *directly connected neighbors* refers to both the neighbors with incoming and outgoing edges.

Plan Exploration: In each iteration of the algorithm, using the same benefit function as in the bidirectional algorithm, we choose the highest ranked data source, $ProbeNode$, from the *center neighbor queue*. We try to add $ProbeNode$ to the existing query plan and see whether the *node connection property* is respected (lines 4-6). If the *node connection property* is violated, we select the second highest ranked node from the queue to repeat the above procedure. Suppose $ProbeNode$ respects the *node connection property*, then $ProbeNode$ and the edge connecting $ProbeNode$ with its neighbor in the existing query plan are added to the query plan (lines 7-9). If there are multiple edges connecting $ProbeNode$ with nodes in the plan, we always select the edge that gives the shortest distance between $ProbeNode$ and the connected aggregation data source (the shortest distance heuristic). After adding

$ProbeNode$ to the query plan, the distance from each node in the plan to any query keyword is updated based on the shortest distance heuristic (line 10-11). After the distance updating phase, if a data source D in the query plan no longer has the shortest distance to any of the keywords, it should be deleted from the plan. Since $ProbeNode$ is added to the query plan, we update the *center neighbor queue* by adding the direct neighbors of $ProbeNode$. If any data source D is deleted from the query plan, D's direct neighbors that are in the *center neighbor queue* should also be removed (lines 12-15).

Algorithm IV.2: Center-spread($SKQQ$)

```
1 Initialization
2 foreach aggregation data source node AN
3   CNQ.add(Neighbors(AN))
4 while CNQ≠ Φ
5   select the node with highest rank from CNQ
6   denot it as ProbeNode
7   if ProbeNode satisfies Node Connection Property
8     add ProbeNode to query plan
9     add Neighbors(ProbeNode) to CNQ
10      foreach keyword∈Q
11        update shortest distance from ProbeNode to keyword
12      foreach node∈query plan
13       if no shortest path to all keywords in Q via node
14         delete node from query plan
15         delete Neighbors(node) from CNQ
16   if find path with finite distance to any keyword
17      from a subset of starting node
18        we find query plan for the query
19 if No nodes in CNQ satisfies Node Connection Property
20   Abort and inform the user
```

Algorithm Termination: When all query keywords can be reached from a subset of the starting nodes with a finite distance, a query plan is obtained (lines 16-18). During the initialization phase, if no data source in the center neighbor queue can be added to the query plan without violating the *node connection property*, the algorithm will conclude that the query cannot be answered completely. If this happens, our system would notify the users, since we cannot generate a correct plan for this query (line 19-20).

D. Query Planning for Nested Entity Subquery: Formulation and Algorithm

Consider the query "*MAX(Asian_Allele_Frequency), Gene* {*Ontology_Function,X*}". The nested entity subquery "{*Ontology_Function, X*}" implies that we want to find the *Genes* that have the same ontology function as X. More specifically, we need to include the outer query entity keyword *Gene* into the nested entity sub-query to make the query meaningful.

Formally, we could consider a nested entity sub-query to have a keyword set $\{b, a, e_1, \ldots, e_k\}$, where, b is the entity keyword in the outer query in which this sub-query is enclosed, a is the original attribute keyword in the nested entity sub-query, and e_1,. . .,e_k are the original entity keywords in the nested entity sub-query. The intent of a nested entity sub-query is to find the entities specified by the keyword b that have the same value on the attribute a as the entities that are

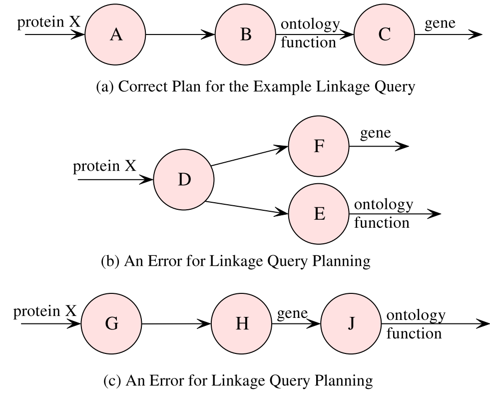


Fig. 8. Query Plan Example for Nested Entity Sub-query (a) Correct Plan for the Example Query, (b) and (c) Plans with Errors

specified by the entity keywords e_1,. . .,e_k. We can say that for a nested entity sub-query, the attribute keyword a *links* the entities specified by b and the entities specified by $e_1, \ldots, e_k$.

As a result, in query plan of a nested entity sub-query, we require the data source covering the linking attribute a (denoted as linking data source) to be topologically before the data source covering the entity b. This is denoted as the *node linking property*. Let us consider the query "*Gene*{*Ontology_Function, Protein X*}". A valid query plan for this query is shown in sub-figure (a) in Figure 8. Using this plan, from the data sources A and B, we can obtain the ontology functions protein X plays. Next, from the data source C, we can obtain the genes that have the same ontology functions. Sub-figure (b) in Figure 8 shows a plan with an error, because the linking data source E is not connected with the data source F covering the outer query entity (gene). As a result, although we can obtain the ontology functions protein X plays and genes related with protein X, but there is no way for us to find the relation between ontology functions and genes. Sub-figure (c) shows another plan with an error, as the linking data source J is topologically after the data source H covering the outer query entity (gene). From this plan, we can only know the ontology function that a gene plays and how this gene is related with the protein X, but we cannot find the expected mapping among the protein X, the ontology function, and the gene, as required in the query.

Formulation for Nested Entity Query Planning: Given a nested entity subquery that has a linking attribute keyword $linkk$ and an outer query entity keyword $outk$, we want to find a minimal (smallest size) sub-graph, $SubG$, as formulated in Section IV-B, which also satisfies the *node linking property*. We have the following result:

Lemma IV.3. *The query planning problem for linkage SKQ is NP-hard.*

Proof. Lemma IV.3 can be proved in a similar fashion as Lemma IV.2. The detailed proof is omitted.

□

Query Planning Algorithm: To ensure the *Node Linking*

Property, we use a variation of the center-spread planning algorithm. In the *linkage planning algorithm*, we consider the linking data source as the *center* data source in the initial query plan. We incrementally add data sources to the query plan to enlarge it. When a data source D that covers the outer query entity keyword is added to the plan, we will check for the *node linking property*. If this property is not respected, the data source D and the data sources in the plan that connect D with the linking data source are removed from the plan. Other parts of the linkage planning algorithm are the same as the center-spread planning algorithm.

V. Query Plan Combination and Merging

In this section, we describe how the complete query plan of a query is obtained by combining and merging the query plans of the sub-queries.

A. Plan Combination

In a query Q, if sub-query $subQ_i$ is the parent of sub-query $subQ_j$, we should combine the plan of $subQ_j$ with the plan of $subQ_i$. We denote the data source nodes in the plan of $subQ_j$ that provide the final output of $subQ_j$ as the *ending nodes*. Similarly, the data source nodes in the plan of $subQ_i$ that need the output from $subQ_j$ as their input are considered the *receiving nodes*. During plan combination, we add plan edges from the *ending nodes* in $subQ_j$ to the *receiving nodes* in $subQ_i$.

In the example in Section I, the only *ending node* of the nested entity queries is `GO` and the only *receiving node* of the main query is `SNP500Cancer`. Thus, we link `GO` with `SNP500Cancer` to obtain the combined query plan.

B. Plan Merging

While plan combination can create a correct query plan, merging components of two query plans can help achieve better efficiency. As shown in the example in Figure 2, there can be data sources in the combined plan that could be merged due to similarity among the sub-queries. The main purpose of plan merging is to reduce the transmission cost of a query plan, where the transmission cost is defined to be the total number of terms transmitted during a query plan's execution [7].

Our planing merging approach is based on an adaptation of an existing planing merging approach developed by Kementsietsidis *et al* [7]. In this approach, a plan edge is denoted as $e = \{n_1, n_2\}$, where n_1 and n_2 are the two ending data sources. Two edges $e_1 = \{n_1, n_2\}$ and $e_2 = \{n_3, n_4\}$ can be merged if the following two conditions hold: 1) the two edges are compatible, i.e., $n_1 = n_3$ and $n_2 = n_4$, and 2) the ordering of the two edges are respected, i.e., if edge e_1 and e_2 are merged, edge e_3 and e_4 are merged, and e_1 topologically proceeds e_3, then e_4 should not topologically proceed e_2.

Given multiple query plans with a list of mergeable edges, Kementsietsidis *et al* used a *partial order alignment* method to find the optimal merging (the merging which reduces the most transmission cost). In the partial order alignment method, a *compatibility graph* $CG = (V, E)$ is built. In CG, each node

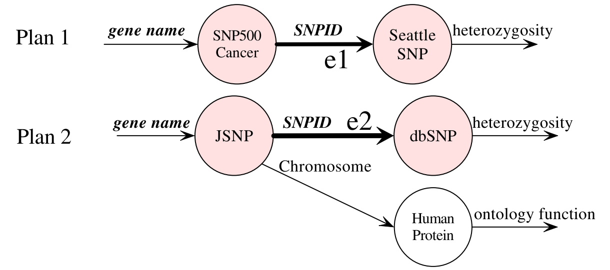


Fig. 9. Example for Modified Merging Condition

n_{CG} represents a pair of mergeable plan edges e_1 and e_2. There is an edge e_{CG} between two nodes $n_{CG1} = (e_1, e_2)$ and $n_{CG2} = (e_3, e_4)$ if the two pairs of mergeable plan edges do not map the same data source node to different data sources in the query plan. Then, the optimal merging can be obtained by finding a maximal clique in the compatibility graph CG. The details of this approach can be found in [7], [22].

In our implementation, we made two modifications to the existing approach: *merging condition modification* and *merging algorithm modification*, which are described below.

Merging Condition Modification: In the existing method, two plan edges $e_1 = \{n_1, n_2\}$ and $e_2 = \{n_3, n_4\}$ are compatible if each pair of ending data sources must be *the same*. Considering different deep web data sources may overlap in terms of certain attributes (*partial redundancy*) [19], we define two edges e_1 and e_2 to be compatible if the *used* input and output attributes of the data sources on the edges are the same. Consider the two edges $e1 = (SNP500Cancer, SeattleSNP)$ and $e2 = (JSNP, dbSNP)$ in Figure 9. If we require mergeable edges to have *exactly the same* ending data sources, edges $e1$ and $e2$ cannot be merged. However, in this example, we have data redundancy on $e1$ and $e2$, i.e., both `SNP500Cancer` and `JSNP` take `gene name` as input and have `SNPID` as output (although `Chromosome` is also an output for `JSNP`, it is irrelevant to edge $e2$), and both `SeattleSNP` and `dbSNP` take `SNPID` as input and have `heterozygosity` as output. As a result, the *used* input and output of the each pair of ending data sources is the same for edge $e1$ and $e2$. Therefore, $e1$ and $e2$ can be merged.

Merging Algorithm Modification: In the existing method, since only *exactly the same* data sources can be merged, the *compatibility graph* is unweighted, i.e., we give no preference to different pairs of mergeable edges. But in our case, different data sources that possibly provide data with different quality, can also be merged. Thus, we need to assign preferences to different mergeable edges.

Mergeable Edges Weight: The merge weight of two mergeable edges $e_1 = (n_1, n_2)$ and $e_2 = (n_3, n_4)$ is $Sim(n_1, n_2) + Sim(n_3, n_4)$, where $Sim(n_1, n_2) = |Benefit(n_1) - Benefit(n_2)|$. $Benefit(n_i)$ is the node score computed by the benefit function used in the query planning algorithm introduced in Section IV-B.

Finding Maximal Edge-weighted Clique: With the above mergeable edge weights, the *compatibility graph* CG in our

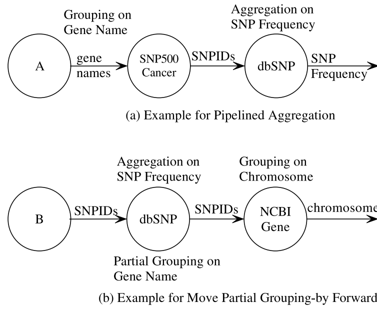


Fig. 10. Example for Illustrating the Two Grouping Optimizations

scenario is an edge weighted graph. Thus, we need to find a maximal edge-weighted clique from the compatibility graph to solve the optimal merging. For this purpose, we adopt the *Reactive Local Search (RLS) Algorithm* [23], which finds maximal cliques by stochastically adding and deleting nodes with the largest edge degree to and from the current clique. We make the following modification to the original algorithm: when adding a node to the current clique, among the largest degree nodes, we prefer the node which can bring in the *largest weighted edge*. Similarly, when deleting a node, among the largest degree nodes, we prefer the node which takes out the *smallest weighted edge*.

VI. Optimization on Query Execution

In this section, we introduce two optimization techniques for query plan execution, which are *pipelined aggregation* and *moving partial grouping-by forward*. Recall in Section IV-C, we defined the *aggregation* and *grouping* attributes in a query and the *aggregation* and *grouping* data source in a query plan.

A. Pipelined Aggregation

We define an aggregation query plan to be *grouping-first* if the grouping data source topologically proceeds the aggregation data source(s). The query plan in sub-figure (a) in Figure 10 is a *grouping-first* query plan. Here, we want to form groups using *gene name* (A is the grouping data source) and then for each group, perform aggregation function on *SNP frequency* (`dbSNP` is the aggregation data source). Suppose we have a gene X which contains 20 SNPs. This optimization will imply that we feed the 20 SNPs to `dbSNP` in a pipelined manner and perform the aggregation incrementally. *Pipelined aggregation* can reduce the query plan transmission cost by *early pruning* if the aggregation is involved in a comparison predicate. For example, suppose we want to find the gene group that has a maximal SNP Frequency smaller than 0.6. If we do not use *pipelined aggregation*, for gene X, we need to issue 20 queries on `dbSNP` to compute the aggregation function, even if the second SNP in X has a frequency value large than 0.6. Using *pipelined aggregation*, we issue only 2 SNP queries, as gene X can be pruned.

B. Move Partial Group-by Forward

We define an aggregation query plan to be *aggregation-first* if the aggregation data source topologically proceeds the grouping data source. The query plan in sub-figure (b) in Figure 10 is an *aggregation-first* query plan, in which we want to form groups using *chromosome* (`NCBI Gene` is the grouping data source). Then, for each group, we perform aggregation function on *SNP frequency* (`dbSNP` is the aggregation data source). Suppose in this example, we have 20 SNPs from 2 chromosomes. Without optimization, we would issue 20 SNP queries on `dbSNP` to find frequency data, next, we will issue 20 SNP queries on `NCBI Gene` to find the 2 chromosomes, and finally, we will perform the group-by and aggregation. There will be a total of 40 queries issued. But, if we know that `dbSNP` also provides the *gene* in which an SNP is located, we could execute this plan in an alternative way. After issuing 20 queries on `dbSNP` to find the frequency and gene data, we first perform a *partial group-by* for SNPs on genes. Suppose, the 20 SNPs are grouped into 4 gene groups, and further, we know that the SNPs from the same gene group must map to the same chromosome. We only need to issue 4 SNP queries on `NCBI Gene` (one SNP per gene group) to obtain the chromosome data. The total plan transmission cost is reduced from 40 to 24. We refer to this technique as *move partial group-by forward*.

The idea of *moving partial grouping-by forward* was originally proposed in relational database query optimization [24]. Specific to the deep web scenario, we have an aggregation SKQ, and its query plan is *aggregation-first*. Suppose the grouping data source is GD, the grouping attribute is ga, the aggregation data source is AD, and the input attribute of AD is aa. We could perform *move partial grouping-by forward* method if all the following conditions hold: 1) the aggregation data source AD covers a term pga, 2) there is a 1 to 1 relationship between the entity specified by pga and the entity specified by ga, and 3) there is a N to 1 relationship between the entity specified by aa and the entity specified by pga.

Under this circumstance, we call the term pga a *partial grouping-by attribute*, and we could do a *partial grouping-by* operation using pga on data source AD for the aggregation attribute aa. In the example above, pga is `Gene Name` and the input of AD (`dbSNP`) (aa) is SNPID. Since a gene contains multiple SNPs (N to 1 relation between SNP and gene), and a gene must be located in one chromosome (1 to 1 relation between gene and chromosome), we can infer that the SNPs locates in one gene must from the same chromosome. As a result, performing *partial group-by* could reduce transmission cost as shown above. The relationship between terms is captured through the domain ontology introduced in Section III.

VII. Domain Ontology and Ranking Benefit Function

In this section, we introduce the domain ontology and benefit functions used in our query planning algorithm.

A. Ontology Overview

Our domain ontology is designed with the following two goals. First, we want to map search terms to data source

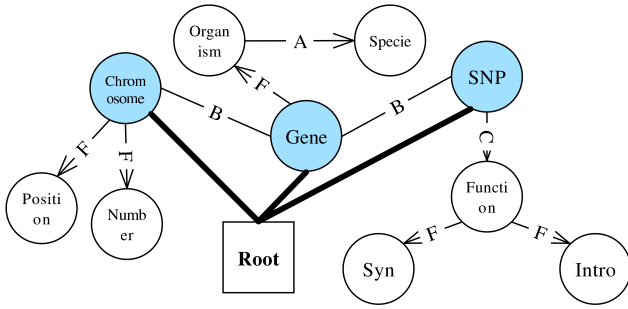


Fig. 11. An Ontology Graph Example

node when search terms query used are alias or synonyms of the attribute names used in data source schemas. Second, ontology information could help us rank similar data sources with respect to the query, and domain ontology is used as a component of the beneficial functions in our query planning algorithm introduced in Section IV-B.

Our ontology is a *schema level* ontology that only contains attribute terms, but not entity names. For example, Gene Name is in our ontology, but ERCC6, a specific gene name, is not. Because the number of attribute terms is likely quite limited in a domain, our ontology remains small and scalable to a large number of data sources.

Our ontology is a connected directed graph $OG = (ON, OE)$. ON is the set of nodes in the ontology graph, and OE is set of edges. The nodes in ontology graph are the domain terms and edges are relations between these terms. Figure 11 shows an ontology graph example, which is specific to our implementation in the biological context.

Domain Term: There are three types of domain terms in the ontology: concept terms, attribute terms, and a root term. Concept terms are high-level conceptual terms, such as Chromosome and Gene for biological domain, which are not among the input or output attributes of any data source. In Figure 11, there are three concept terms (shaded nodes) and seven attribute terms (unshaded nodes).

Ontology Relation: We define four types of ontology relations. 1). *A type relation.* It connects a term with its synonym (Organism and Specie in the Figure 11). 2). *B type relation.* It connects two concept terms, which are related in the domain a particular implementation is targetting. 3). *C type relation.* It captures the class-subclass relationship. In the Figure 11, SNP can be categorized into two sub-types using the Function attribute. 4). *F type relation.* It connects a biological concept term with its attribute terms or connects a high-level attribute term with its low-level attribute terms. In the Figure 11, position and number are two attributes of Chromosome indicated by two F relations. Among the four types of relations, A and B types are undirected and C and F types are directed, as shown in Figure 11.

B. Data Source Representatives

Within our implementation that targets a specific domain, each data source is designed to provide information of some specific sub-domain(s) within that domain. For example, Seattle provides Human SNP data and Entrez Gene provides gene and protein related data. In our system, for a data source D, we define the concept terms, which correspond to the sub-domains that D covers, as the *representatives* of D. The representative of Seattle is SNP, and the representatives of Entrez Gene are Gene and Protein. We can compute the representatives of a data source D on the ontology graph as follows: if we reverse-traverse the ontology graph starting from all nodes corresponding to D's output attributes through the F or C links, all the concept terms reached are the representatives of D. Suppose D has n representatives $r_1, r_2, \ldots, r_n$, then each representative r_i is associated with a weight w_i, which is the percentage of D's output attributes which finally reach r_i.

The representatives associated with their weights can be used to estimate the relevance of a data source w.r.t. a query keyword. The larger the weight w_i of representative r_i, the more output attributes of D are related to r_i, and therefore, we believe D is more likely to be focusing on providing information about r_i with high quality. Suppose we have two data sources A and B, in Figure 11, A has Position, Number and Organism as output attributes, and B has Number and Function as outputs. Following the F and C links, we know the representatives of A are Chromosome and Gene with weights of $\frac{2}{3}$ and $\frac{1}{3}$ respectively, and the representatives of B are SNP and Chromosome with weights of $\frac{1}{2}$ and $\frac{1}{2}$ respectively. If a query keyword is related to the term Chromosome, we would predict A is more relevant to the keyword than B given all other properties of A and B are equal. The detail usage of data source representatives as a relevance measure is introduced in Section VII-D.

C. Ranking Strategy Overview

We have two types of ranking function, *node ranking function* and *edge ranking function*. The node ranking function should give a data source higher node score if the node has the following properties: 1) it can cover more search terms, 2) it is closer from the set of starting nodes, because we prefer a query plan with smaller size, 3) it provides the data with higher quality and 4) it satisfies the constraints which are specified in the query. Similarly, an edge has shorter distance if the exploration of this edge can help to narrow down the search space or provide more accurate answer to a query. Since the second type of dependence relation defined in Section III-B is used for this purpose, we favor the edges with the second type of dependence over the edges with the first type of dependency only.

D. Node Ranking Strategy

Given a node n_i, we first define the node score of n_i w.r.t. search term k_j as $NScore(n_i, k_j) = c_{ij} \times q_{ij}$. Here, c_{ij} is the node coverage score, and q_{ij} is the node quality score. We further define the node score of node n_i w.r.t. the entire query Q as $NScore(n_i) = \frac{\sum_{k_j \in Q} NScore(n_i,k_j)}{m} \times constraint_{n_i}$, where m is the total number of search terms in the query,

and $constraint_{n_i}$ is the node constraint match score of n_i w.r.t. query Q.

Node Coverage Score c_{ij}: We want the node that covers more search terms and is closer to the starting nodes to be ranked higher. Specifically, we defined $c_{ij} = \frac{1}{Level(i)+1} \times Is_Contain(j)$. $Level(i)$ is the shortest distance in terms of the number of edges from any of the starting nodes to node n_i. $Is_Contain(j)$ is a function that returns 0 if data source n_i does not contain k_j as its output, returns 1 otherwise.

Node Quality Score q_{ij}: q_{ij} measures the relevance of n_i w.r.t. k_j. Intuitively, as explained in Section VII-B, the shorter the distance from k_j to n_i's representatives, the higher the relevance. To compute the relevance score, we follow three steps. First, we find the representatives of data source n_i, as $r_1^i, r_2^i, \ldots, r_m^i$ with weights $w_1^i, w_2^i, \ldots, w_m^i$. Then, we compute similarity between term k_j and representative r_l^i using an ontology based similarity metric defined in [25]. Denoted it as $Sim(r_l^i, k_j)$. The final node quality score is the weighted sum of all relevance scores of each representative w.r.t. keyword k_j, which is denoted as $q_{ij} = \sum_{l=1}^{m} w_l^i \times Sim(r_l^i, k_j)$.

Node Constraint Match Score $constraint_{n_i}$: Some search terms represent constraints that a user sets on the query. Each data source has a C attribute which represents its inherent constraints. We set $constraint_{n_i} = -\infty$ if data source n_i has conflicting constraints with the query, and $constraint_{n_i} = \frac{MC+\epsilon}{SC+\epsilon}$, if there is no conflicting constraints. MC refers to the number of *matched constraints* and SC refers to the number of *superfluous constraints*. ϵ is a small positive number used to avoid 0 in both numerator and denominator. We prefer the data source that has a high ratio of matched constraints. We emphasize that superfluous constraints lead to *incomplete* answers but not *wrong* answers. For example, a query wants `SNPIDs` of `ERCC6`, and a data source providing `SNPID` has constraints `Het` > 0.2 which means that it only has `SNPIDs` whose `Het` value is greater than 0.2. Using this data source will obtain a subset of the desired answers, but not wrong answers.

E. Edge Ranking Strategy

The edge score is considered as the distance of the edge from one node u to its descendant v. Based on the desired properties of edge score as in Section VII-C, the edge score of e connecting u and v is defined as $EScore(u, v) = \frac{1}{num_2+1}$, where num_2 is the number of second type dependency on edge (u,v). A larger value of num_2 indicates shorter distance edge, thus a higher ranked edge.

VIII. Evaluation

This section describes the experiments we conducted to evaluate our approach.

A. Experiment Setup

Our evaluation was done using 12 biological deep web data sources which include `NCBI dbSNP`[1], `NCBI Gene`[1], `NCBI Protein`[1], `NCBI BLAST`[1], `SNP500`[2], `Seattle`[3], `SIFT`[4], `BIND`[5], `HGNC`[7], `ALFRED`[8], `Human Protein`[9], and `Uniprot`[10]. The input and output schema of the data sources were extracted using a previously created wrapper. The dependency graph of the data sources was constructed by analyzing the correspondence between the output and input attributes of different data sources. The queries we use for evaluation throughout this section are based on our collaboration with a biologist focusing on SNP-related studies [26].

Our evaluation study has two components. The first components evaluates the performance of the query planning algorithms, with focus on scalability and comparison with an optimal algorithm. The second part illustrates the effectiveness of the merging and grouping optimization techniques proposed in Sections V and VI.

B. Query Planning Algorithm Evaluation

First, we evaluate the performance of our heuristic query planning algorithm. Second, we will examine the scalability of the query planning algorithm shown in Algorithm IV.1. Finally, we analyze the overhead of the query planning algorithm.

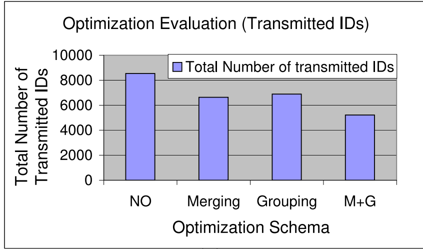
As we stated in Section IV, the query planning problems we have considered are NP-hard, and our algorithms are heuristic in nature. Here, we show the performance of our heuristic planning algorithms. For each query, we first generate a plan using our heuristic planning algorithms, and then we generate an optimal plan using an exhaustive search algorithm. The comparison between the query plans is shown in Table III.

We can observe that, for 18 out of 20 cases, the size of the plan, measured as the number of nodes in subgraph, from our heuristic algorithms is exactly the same as the size of the optimal plan. At the same time, the query planning times of the heuristic algorithms are faster by at least two orders of magnitude. This shows our heuristic algorithms are very effective. Besides the size, we also compared the actual results extracted by the plans generated from both algorithms, and the plans from our heuristic algorithm can always extract the same results as the optimal plans.

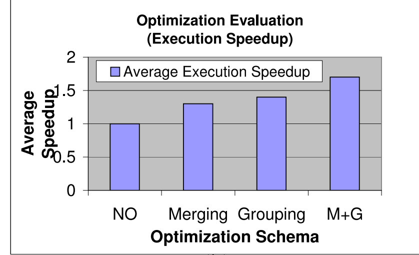
We next evaluate the scalability. In Figure 12, the x-axis is the number of sub-queries used. The y-axis is the query planning time (in milli-seconds) of the planning algorithm, normalized in a logarithmic scale. We observe that with the increase in the number of sub-queries, the query planning time increases in a linear fashion. This shows that our planning algorithms have good scalability. While measuring the planning overhead, we found that for 50 queries, query planning time is, on the average, only 0.03% of the query execution time.

[1]http://www.ncbi.nlm.nih.gov/

[2]http://snp500cancer.nci.nih.gov/home_1.cfm
[3]http://pga.gs.washington.edu/
[4]http://blocks.fhcrc.org/sift/SIFT.html
[5]http://www.bind.ca
[7]www.genenames.org
[8]http://alfred.med.yale.edu/alfred/
[9]http://alfred.med.yale.edu/alfred/
[10]http://www.uniprot.org/



(a)



(b)

Fig. 13. Impact of Optimizations (a) Total Number of Transmitted Terms, (b) Query Execution Speedup

TABLE III
NUMBER OF DATA SOURCES IN QUERY PLAN COMPARISON

Query ID	# of DS optimal	# of DS heuristic	Query ID	# of DS optimal	# of DS heuristic
1	2	2	11	8	8
2	4	4	12	5	5
3	2	2	13	6	6
4	3	3	14	2	2
5	**6**	**7**	15	6	6
6	4	4	16	6	6
7	**5**	**6**	17	8	8
8	4	4	18	9	9
9	6	6	19	7	7
10	8	8	20	4	4

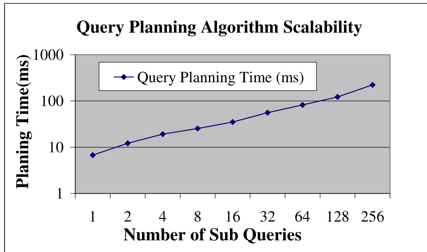


Fig. 12. Planning Algorithm Scalability

C. Optimization Techniques Evaluation

In this section, we evaluate the performance of the proposed optimization techniques.

In this experiment, we consider four different schemes, which are No Optimization (NO), Only Merging Optimization (Merging), Only Grouping Optimizations (Grouping) and Merging+Grouping Optimizations (M+G). We used 50 queries in this experiment. We execute the queries using all four schemes. We focus on two metrics, which are the *total number of terms transmitted* during the execution and the *query execution time*. Irrespective of the scheme used, all query plans are executed in a *parallel fashion*, i.e., *independent data sources* are accessed in parallel.

The results are shown in Figure 13. The x-axis in both sub-figures shows the scheme used. The y-axis in sub-figure (a) shows the total number of terms transmitted, whereas in sub-figure (b), it shows the speedup of the query execution time normalized with respect to the NO version. From sub-figure (a), we observe that using the merging and grouping optimizations independently, the total number of transmitted terms is decreased by about 25% and 20% , respectively. When they are combined, we achieved a 40% decrease in the number of transmitted terms. From sub-figure (b), we observe a similar trend. The execution speedups achieved using Merging and Grouping independently are 1.3 and 1.4, respectively. We achieved a 1.7 time speedup when we combined them. We can see that although the merging optimization reduces more transmitted terms, as compared with the grouping, the execution time speedup of merging is lower than that of grouping. This is because the merged data sources should be executed in a synchronized manner, which introduces synchronization costs.

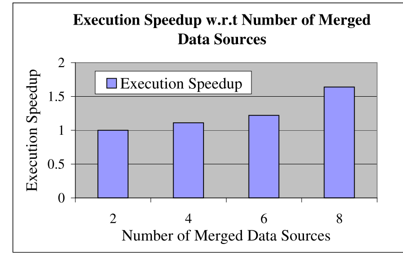


Fig. 14. Execution Time Speedup Analysis w.r.t. Number of Merged Data Sources

For the only merging optimization schema, we compare the execution time speedup with respect to the number of data sources being merged. The result is shown in Figure 14. The x-axis is the number of data sources merged , and the y-axis is the execution time speedup normalized with respect to No Optimization. We can observe that with the increase in the number of merged data sources, the execution speedup also increases, as one would expect.

IX. RELATED WORK

We now compare our work with existing work on related topics, including query processing and optimization for web data sources, query mediation systems, keyword search on relational databases, and deep web integration and crawling systems.

Query Processing and optimization on Web Data Sources: Query processing and optimization on web data sources has

been an important topic in recent years [7], [5], [6], [27], [28], [29]. Kementsietsidis *et al* [7] have developed a system for optimizing exploratory queries over biological data sources. The main optimization they consider is merging data sources among related exploratory queries. As we had explained earlier, the merge algorithm in our work is closely based on their work. Srivastava *et al* [6] presented an algorithm for ordering data sources in a query plan with the goal of minimizing the query's execution time. However, the algorithm is based on the assumption that only one attribute is provided by a single data source. Several other query planning algorithms for web data sources have also been proposed [5], [27], [28]. None of the above work considers query planning and optimization for complex queries, i.e., including aggregation, group-by, and nested queries, which is the focus of our work. Two of these efforts are based on the *Steiner Tree* algorithm [27], [28], where it is assumed that a valid query plan must be a *tree*. In comparison, query plans in our case can be disconnected directed acyclic graphs.

Query Mediation Systems: Use of mediators is one of the classical approaches for information integration. There have been several well-developed systems in this area, include SIMS [9], Information Manifold [10], TSIMMIS [11], and MedMaker [12], and a bioinformatics mediator platform developed by Raschid *et al* [30]. A mediator provides users with seamless integrated views of the data from heterogeneous sources and is capable of generating query plans for user queries. But query plans for the mediator system have been generated based on pre-specified rules or axioms. For example, the Mediator Specification (MS) rules are used in TSIMMIS and MedMaker, and the SIMS Axioms are used in SIMS. There are two key differences in our approach. First, our system can handle more complex queries, including the nested and aggregation queries. Second, our data source model and the cost metric are very distinct, and as a result, the query planning formulation and algorithms are different.

Keyword Search on Relational Databases: Recently, keyword search over relational databases has attracted a lot of attention [13], [14], [21], [16], [17]. In relational database keyword search, data tuples are represented as nodes, and foreign keys are represented as edges. In our case, since we do not have the access to the database behind deep web data sources, our graph model is at the *metadata* level. Furthermore, except for the work by Tata *et al.* [17] (SOAK system), the work on relational database keyword search does not consider aggregation queries, and infact, none of them has considered nested queries.

Deep Web Crawling and Integration Systems: Lately, there has also been much effort on mining useful information from the deep web [1], [4], [31]. This work has been focused on database integration, schema matching, and hidden data crawling. The focus of our work is distinct, i.e. answering complex cross-source queries over multiple inter-dependent deep web data sources. Our work assumes that a set of relevant deep web sources have been found and integrated for a particular domain.

X. CONCLUSION

This paper has considered the problem of executing complex queries over a set of integrated deep web data sources. We have proposed a structured keyword query approach, in which a user could easily formulate a complex query, including aggregation, group-by, value constraints, and nested queries. We have formulated the query planning problem for such queries over the deep web, with the main consideration being minimizing the number of geographically distributed data sources used in answering a query. We have also considered several optimizations for reducing the query execution time. Our experiments show that our query planning algorithms are efficient and effective, and the optimization techniques can speedup the query execution by an average factor of 1.7.

REFERENCES

[1] B. He, Z. Zhang, and K. C.-C. Chang, "Knocking the door to the deep web: Integrating web query interfaces," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of Data*, 2004, pp. 913–914.

[2] K. C.-C. Chang and J. Cho, "Accessing the web: From search to integration," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of Data*, 2006, pp. 804–805.

[3] K. Chang, B. He, and Z. Zhang, "Toward Large Scale Integration: Building a Metaquerier over Databases on the Web," 2005.

[4] H. He, W. Meng, C. Yu, and Z. Wu, "Automatic integration of web search interfaces with wise_integrator," *The international Journal on Very Large Data Bases*, vol. 12, pp. 256–273, 2004.

[5] D. Braga, S. Ceri, F. Daniel, and D. Martinenghi, "Optimization of Multi-domain Queries on the Web," *Proceedings of the VLDB Endowment*, vol. 1, pp. 562–673, 2008.

[6] U. Srivastava, K. Munagala, J. Widom, and R. Motwani, "Query optimization over web services," in *Proceedings of the 32nd international conference on Very Large Data Bases*, 2006, pp. 355–366.

[7] A. Kementsietsidis, F. Neven, D. V. de Craen, and S. Vansummeren, "Scalable multi-query optimization for exploratory queries over federated scientific databases," *Proceedings of the VLDB Endowment*, vol. 1, pp. 16–27, 2008.

[8] F. Wang, G. Agrawal, and R. Jin, "Query planning for searching inter-dependent deep-web databases," in *Proceedings of the 20th international conference on Scientific and Statistical Database Management*, 2008, pp. 24–41.

[9] Y. Arens, C. A. Knoblock, and W.-M. Shen, "Query Reformulation for Dynamic Information Integration," *Journal of Intelligent Information Systems - Special Issue on Intelligent Information Integration*, vol. 6, no. 2/3, pp. 99–130, 1996.

[10] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava, "The Information Manifold," in *Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, 1995, pp. 85–91.

[11] H. Garcia-molina, Y. Papakonstantinou, D. Quass, Y. Sagiv, J. D. Ullman, V. Vassalos, and J. Widom, "The TSIMMIS Approach to Mediation: Data Models and Languages," *Journal of Intelligent Information Systems*, vol. 8, pp. 117–132, 1997.

[12] Y. Papakonstantinou, H. Garcia-molina, and J. Ullman, "Medmaker: A mediation system based on declarative specifications," in *Internation Conference on Data Engineering*, 1996, pp. 132–141.

[13] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases," in *Proceedings of the 28th international conference on Very Large Data Bases*, 2002, pp. 67–681.

[14] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "Efficient ir-style keyword search over reltional databases," in *Proceedings of the 29th international conference on Very Large Data Bases*, 2003.

[15] B.Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, P. Parag, and S.Sudarshan, "Banks: Browsing and keyword searching in relational databases," in *Proceedings of the 28th International Conference on Very Large Data Bases*, vol. 28, 2002, pp. 1083–1086.

[16] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "Ease: An effective 3-in-1 keyword search method for unstructured, semi-structured and structured data," in *Proceedings of the 2008 ACM SIGMOD international conference*, 2008, pp. 903–914.
[17] S. Tata and G. M. Lohman, "Soak: Doing more with keywords," in *Proceedings of the 2008 ACM SIGMOD*, 2008, pp. 889–901.
[18] A.J.Brookes, "The Essence of SNPs," *Gene*, vol. 234, pp. 177–186, 1999.
[19] M. K. Bergman, "The Deep Web: Surfacing Hidden Value," *Journal of Electronic Publishing*, vol. 7, 2001.
[20] H. Davulcu, J. Freire, M. Kifer, and I. Ramakrishnan, "A layered architecture for query dynamic web content," in *Proceedings of the 1999 SIGMOD Conference*, 1999, pp. 491–502.
[21] V. Kacholia, S. Pandit, S. Chakrabarti, S.Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases," in *Proceedings of the 31st international conference on Very Large Data Bases*, 2005, pp. 505–516.
[22] N. Moreano, G. Araujo, Z. Huang, and S. Malik, "Datapath Merging and Interconnection Sharing for Reconfigurable Architecture," in *Proceedings of the 15th international symposium on System Synthesis*, 2002, pp. 38–43.
[23] R. Battiti and M. Protasi, "Reactive Local Search for the Maximum Clique Problem," *ACM Journal of Experimental Algorithmics*, vol. 2, 1997.
[24] S. Chaudhuri and K. Shim, "Including Group-By in Query Optimization," in *Proceedings of the 20th international conference on very large data bases*, 1994, pp. 354–366.
[25] P. Cimiano, G. Ladwig, and S. Staab, "Gimme' the context: Context-driven automatic semantic annotation with c-pankow," in *Proceedings of the 14th international conference on World Wide Web*, 2005, pp. 332–341.
[26] F. Wang, G. Agrawal, R. Jin, and H. Piontkivska, "Snpminer: A domain-specific deep web mining tool," in *Proceedings of BIBE 2007*, 2007.
[27] P. P. Talukdar, M. Jacob, M. S. Mehmood, K. Crammer, Z. G. Ives, F. Pereira, and S. Guha, "Learning to create data-integrating queries," *Proceedings of the VLDB Endowment*, vol. 1, pp. 785–796, 2008.
[28] R. Varadarajan, V. Hristidis, and L. Raschid, "Explaining and reformulating authority flow queries," in *Proceedings of the 2008 IEEE ICDE international conference*, 2008, pp. 883–892.
[29] A. Cali and D. Martinenghi, "Querying Data under Access Limitations," in *Proceedings of the 24th International Conference on Data Engineering*, 2008, pp. 50–59.
[30] B. A. Eckman, T. Gaasterland, Z. Lacroix, L. Raschid, B. Snyder, and M. E. Vidal, "Implementing a bioinformatics pipeline (bip) on a mediator platform: Comparing cost and quality of alternate choices," in *Proceedings of the 22nd International Conference on Data Engineering Workshops*, 2006, p. 67.
[31] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy, "Google's Deep Web Crawl," *VLDB Endowment*, vol. 1, pp. 1241–1252, 2008.