16 Keyword Search: Integration on Demand

Traditionally, data integration is used to either build applications (whether Web-based or more traditional) that provide cross-source information access or to build ad hoc query interfaces for data exploration by relatively sophisticated users. However, a more recent focus has been to enable "average" (non-database-expert) users to pose ad hoc queries over structured, integrated data via a familiar interface: keyword search. However, keyword search in this model is a good deal more complex than in a typical information retrieval system or search engine: it does not merely match against single documents or objects. Intuitively, the set of keyword terms describes a set of concepts in which the user is interested. The data integration system is tasked with the job of finding a way of relating the tables or tuples relating to these concepts, e.g., through a series of joins.

In this chapter, we first describe the abstract problem of keyword search over structured data in Section 16.1, then some popular algorithms for returning ranked results in Section 16.2. Finally, we describe some of the issues in implementing keyword search over data integration applications in Section 16.3.

16.1 Keyword Search over Structured Data

In information retrieval, keyword search involves finding a *single* document with matches to all of the keywords. In a relational or XML setting, the goal is typically to find *different* data items matching the different keywords and to return ways these matches can be *combined* to form an answer.

The general approach to answering keyword queries over structured data sources is to represent the databases as a *data graph* relating data and/or metadata items. Nodes represent attribute values and in some cases metadata items such as attribute labels or relations. Directed edges represent conceptual links between the nodes, where the links in a traditional DBMS include foreign key, containment, and "instance-of" relationships.

A query consists of a set of terms. Each term gets matched against the nodes in the graph, and the highest-scoring trees (with leaf nodes matching the search terms) are returned as answers. A variety of scoring or ranking models have been developed, with slightly different semantics and different levels of efficiency of evaluation.

Principles of Data Integration, DOI: 10.1016/B978-0-12-416044-6.00016-8

We now describe the data graph, scoring models, and basic algorithms for computing results with top scores in more detail. The basic model and many of the techniques apply equally to a single-database setting or a data integration setting. Moreover, with minor variations the same techniques can be applied not only to relational databases, but also to XML or RDF (or even object, network, or hierarchical) databases.

16.1.1 Data Graph

A variety of different graph representations have been proposed for representing the database(s). In the most general form, we have a graph with weighted nodes and weighted directed edges. Nodes may represent specific attribute values within tuples, composite objects such as sets or XML trees, attribute labels, and even relations. In some cases these nodes may be annotated with weights representing authoritativeness, trust, etc.

Directed edges in the graph model the relationships among nodes. Some examples include containment of a value node within a node representing a collection or a tuple; "instance-of" relationships between a label and a data value; foreign key-to-key references; or even relationships based on similarity. In general, these directed edges may also be weighted, indicating the strength of the relationship.

Note that this graph is typically a logical construct used for defining the semantics of query answering; for efficiency reasons it is generally computed lazily.

Example 16.1

An example of a data graph that includes schema components appears in Figure 16.1. This graph represents an excerpt of a bioinformatics setting with four tables (focusing on gene-related terms, protein-related entries, and protein-related publications), indicated as rounded

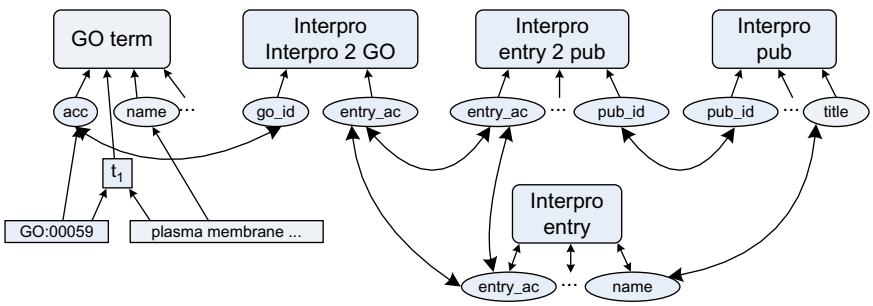

FIGURE 16.1 Example schema/data graph for a set of relations in the bioinformatics domain. This example includes nodes representing relations (rounded rectangles), attribute labels (ellipses), tuples (t_1), and attribute values (rectangles); and edges representing membership (small directed arrows) as well as foreign key or predicted cross-reference (bidirectional arrows).

rectangles. We show the attribute labels for each table (ellipses), member tuples (rectangles such as node t_1), and attribute values (rectangles with text labels). Edges in this example include foreign keys (shown as bidirectional thick arrows), as well as membership (small directed arrows). In many settings, both edges and nodes might be annotated with weights.

A natural question is how to set the weights in a data graph. In general, the weights on data or metadata nodes represent authoritativeness, reliability, accuracy, or trustworthiness. Typically the weights on edges represent similarity or relatedness.

Node weights are generally assigned using one of three approaches:

- **Graph random-walk algorithms.** Inspired by link analysis algorithms such as PageRank, some systems look at the edge connectivity in the graph to assign a score to each node. A popular algorithm here is called ObjectRank, which is a modification of PageRank to a linked-object setting.
- **Voting or expert rating.** In certain settings, especially in scientific data sharing, there may be an advisory board who assigns scores to certain nodes. Alternatively, it is possible to have the entire user community vote on authoritativeness.
- **Query answer-based feedback.** Later in this chapter we discuss an approach whereby the system takes feedback on the quality of specific *query results* to learn authoritativeness assignments to nodes.

In a single-database context, edge weights are typically assigned based on known relationships such as integrity constraints (specifically foreign keys). In the more general data integration setting, edges may need to be inferred, as we discuss in Section 16.3.

16.1.2 Keyword Matching and Scoring Models

Given a set of keyword terms and a data graph, a keyword search system will match each keyword against nodes in the graph and compute a similarity score or weight. We can model this as adding a node for each keyword into the graph, and then adding weighted edges to each (possibly approximately) matching node in the data graph.

Example 16.2

Figure 16.2 shows an example of a *query graph* where nodes and similarity edges have been added into the data graph of Figure 16.1. The keyword nodes are represented in italics, with dashed directed edges to the matching nodes in the data graph. In general, each keyword node may have edges going to multiple nodes in the data graph, and each edge might be annotated with a similarity score or other weight.

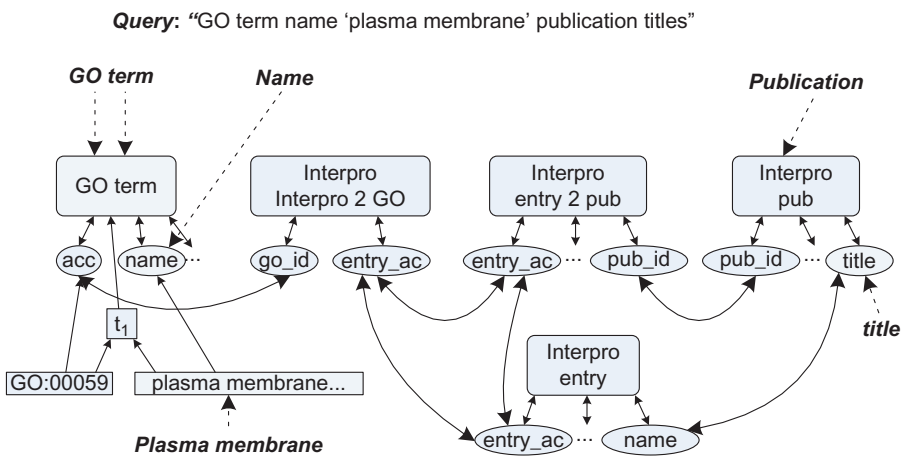

FIGURE 16.2 Example of matching keywords against the schema/data graph.

Under this model, the actual query processing computation finds the sets of trees from the data graph — with keyword nodes as leaf nodes — and returns the top-scoring trees. We describe two of the more popular scoring models next.

SCORE AS SUM OF WEIGHTS IN THE QUERY TREE

One popular approach is to assume that the weights within the tree, including the weights on edges between keyword nodes and data or metadata nodes, are effectively costs or prestige scores, where higher values represent greater distance or less value. This model has a probabilistic interpretation: if each weight represents the negative log likelihood of the probability of its correctness and/or relevance, then the sum of the weights of all of the edges in the tree represents the overall negative log likelihood of tree's correctness or relevance, under an assumption of probabilistic independence. If we only have weights on the edges, then the k highest scoring Steiner trees in the graph represent the k best answers. (A Steiner tree is a tree whose edge weights sum to the smallest value, which connects a designated set of leaf nodes.)

If nodes do have weights, then one option is to simply "convert" these into edge weights in a modified graph, as follows: take each weighted node n_i and split it into nodes n_i and n_i', then add an edge from n_i to n_i' with weight equal to the original $weight(n_i)$ times some scale factor α. Then, as before, a Steiner tree algorithm run over the modified graph will return the top answers.

In certain situations, the query tree is considered to have a *root* (which takes into account the directionality of the edges), and the root may itself be given a score that needs to be scaled and added to the tree score. Again, this can be incorporated into the Steiner tree algorithm: add one additional leaf node R to the modified graph and an edge between

R and each potential root node, with a weight equal to the score the node would receive as root. Then add node *R* to the set of leaf nodes in the Steiner tree.

SCORE AS SUM OF ROOT-LEAF COSTS
The previous scoring model takes into account sharing among edges: each edge's weight is only counted once. Moreover, a root node may be shared across multiple result trees. An alternative scoring model assumes that there should only be one answer per candidate root node and that the score of a tree rooted at node *R* should be based on the sum of the costs of the shortest paths between *R* and the closest node matching each keyword term. Under this model, shared edges are counted multiple times, but there is a computational advantage that each path can be computed separately. As with the previous model, it is straightforward to also incorporate weights on the graph nodes and an additional weight assignment when a node becomes the root.

16.2 Computing Ranked Results

Given a method for assigning scores to possible matches of a keyword query against a data graph, the next challenge is to compute the top-*k*-ranking answers. Computing top-*k* answers is a heavily studied topic in the database research literature, so new techniques continue to be developed. We briefly sketch some of the common techniques used here and refer the reader to the bibliographic references for a more comprehensive treatment.

16.2.1 Graph Expansion Algorithms

As described previously, a query result in the keyword search setting is derived from a Steiner tree over a graph. In some systems, the graph represents the entire database and the Steiner tree is itself an answer. In other systems, the Steiner tree is over the schema (or a combination of schema and data), and thus the tree represents a particular join query to be executed. In either case, Steiner tree computation is an NP-hard problem, hence it is not feasible to use exact Steiner tree algorithms to compute top-scoring results over a large graph.

Instead, the most popular means of computing answers is that of heuristics-based graph expansion, which generally consists of finding a spanning tree and then iteratively refining it. Often the assumption is that the data are to be returned as a tree with a root: this tree's structure is based on the directionality of key-foreign key relationships.

BACKWARDS EXPANSION
One popular scheme, called *backwards expansion*, is to build a complete graph over the terminal (leaf) nodes as follows. Match the search keywords against the set of nodes using an inverted index or similar structure: each match becomes a leaf node. From each such leaf node, create a cluster for the single-source shortest path and populate it with the leaf node. Expand each cluster by following foreign-key edges backwards from existing nodes

in the cluster, according to a lowest-cost-first strategy (using any of the cost models of Section 16.1). Ultimately, the clusters will intersect, meaning that the algorithm has found intersecting paths forming a spanning tree. Such paths can be returned in increasing order of cost.

BIDIRECTIONAL EXPANSION
Sometimes backward expansion can be very expensive, especially if most nodes have high degree, so *bidirectional expansion* refines this. Here, the expansion of clusters is done in a prioritized way. It can be performed in the forward direction from candidate root nodes, as well as through backwards expansion from the leaves. Bidirectional expansion uses a heuristic called *spreading activation*, which prioritizes nodes of low degree and edges with low weights.

COST-BASED BACKWARDS EXPANSION
This technique improves on backwards expansion but requires a cost model where the score is a sum of root-leaf costs as opposed to a sum of edge weights in the query tree (see Section 16.1, and recall that the former model will "double-count" edges shared by shortest paths between the root and multiple leaf nodes). As before, exploration is done in terms of clusters that orginate from leaf nodes. The choice of which cluster to expand next depends on the cardinality of the cluster: the cluster with the fewest items is the one to be expanded, and the nearest node to the cluster (by the cost model) is added to that cluster. The algorithm's running times are further improved by partitioning the graph into blocks and preindexing each block.

Generally, the nodes in the graph represent values or tuples within relational tables, and multiple rows within the same table may match the same keyword. Thus it is often advantageous to "batch" the computation of results as a series of select-project-join queries to compute sets of partial results simultaneously. Such subresults are typically termed *candidate networks*. Results from candidate networks are typically combined using threshold-based merging techniques, discussed next.

16.2.2 Threshold-Based Merging

A key technique in providing the top-scoring query answers — so-called top-k query processing — is that of merging several streams of partial results. Most of the algorithms for doing this are derived from the Threshold Algorithm (TA).

In our context, the goal is to rank a set of tuples where the score of every tuple is computed as a function of multiple base scores. For example, we may be ranking restaurants based on a combined score of their rating and their price range.

For each of the base scoring functions x_i, we have an index L_i through which we can access the tuples in nonincreasing order of the value of x_i. Furthermore, the combined scoring function $t(x_1, \ldots, x_m)$ is monotone, i.e., $t(x_1, \ldots, x_m) \leq t(x'_1, \ldots, x'_m)$ whenever $x_i \leq x'_i$ for every i. Moreover, assume that given a value of x_i for the ith index, we can access all the tuples that contain x_i via random access.

As its name suggests, the Threshold Algorithm is based on the notion of the threshold, which refers to the maximum score of any tuple we have not yet encountered by reading from the various indices. Once we have encountered k tuples whose scores are higher than the threshold, we know that we do not need to read further to return the top-k-scoring answers. At initialization the threshold is set to a maximum value. Each time we read a tuple from one of the indices L_i, this either keeps the threshold the same (if x_i is the same as in the previous tuple read from L_i) or lowers it (if x_i is smaller).

In this context, the Threshold Algorithm can be expressed as follows.

1. In parallel, read each of the m indices $L_1, \ldots, L_m$. As a tuple R is retrieved from one of the lists, do random access and any required joins to produce the set of tuples $\mathcal{R}$ that have the same value for x_i on R. For each $R' \in \mathcal{R}$, compute the score $t(R')$. If R' is one of the k highest combined scores we have seen, then remember R' and $t(R')$ (breaking ties arbitrarily).
2. For each index L_i, let $\underline{x_i}$ be the lowest value of x_i sequentially read from the index. Define a *threshold value* τ to be $t(\underline{x_1}, \underline{x_2}, \ldots, \underline{x_m})$. Note that none of the tuples that have not been seen by the algorithm yet can have a combined score of more than τ. While we have not seen k objects whose score is at least equal to τ, the algorithm continues reading from the indices (and as a result, τ will decrease). Once k objects have been seen whose score is at least equal to τ, halt and return the k highest-scoring tuples that have been remembered.

We can see this more formally specified in Algorithm 14.

Algorithm 14. Threshold Algorithm (TA)

Input: table-valued input T; indices over T, each in descending order of associated score attribute x_i: $L_1 \ldots L_i \ldots L_m$; number of tuples k; cost function t. **Output**: top-k answers.

```
Create priority queue Q of size k
repeat
  for all i := 1...m in parallel do
    Read R_i := next entry in index L_i
    Let x_i be the value of the scoring attribute for R_i
    Let T_i := retrieve(R_i)
    {Retrieve tuples from T indexed by R_i}
    for all T_i ∈ T_i do
      Compute score t(T_i)
      Enqueue T_i into Q; if Q is already full, discard lowest-scoring result, which might be T_i
    end for
  end for
  Set threshold τ = t(x_1, ..., x_m)
until k tuples in Q score above τ
Dequeue and output all elements of Q
```

Many variations of this core algorithm have been proposed, including approximation algorithms and algorithms that do not require random access over the source data (if joins are not required). Moreover, a wide variety of "ranked join" algorithms have been developed using the core ideas of the Threshold Algorithm.

Example 16.3

Suppose we are given a relation *Restaurants* with two scoring attributes, *rating* and *price*, as shown in Table 16.1. Each restaurant (we shall use its name as its ID) has a rating on a scale of 1 to 5 (where 5 is better) and a price on a scale of 1 to 5 (where 5 is most expensive).

Now assume a user wants to find the most promising restaurants and has decided that the rating should be based on a scoring function $score \cdot 0.5 + (5 - price) \cdot 0.5$. If we wish to use the Threshold Algorithm to efficiently retrieve the top three restaurants, we would proceed as follows. First, construct two indices over the table, as shown in Table 16.2: one index is based on the ratings, in decreasing order; the other index is based on the value $(5 - price)$, also in decreasing order.

Table 16.1 Example Restaurants, Rated and Priced

Name	Location	Rating	Price
Alma de Cuba	1523 Walnut St.	4	3
Moshulu	401 S. Columbus Blvd.	4	4
Sotto Varalli	231 S. Broad St.	3.5	3
McGillin's	1310 Drury St.	4	2
Di Nardo's Seafood	312 Race St.	3	2

Table 16.2 Score Attribute Indices over Restaurants

Rating	Name
4	Alma de Cuba
4	Moshulu
4	McGillin's
3.5	Sotto Varalli
3	Di Nardo's Seafood

(a) Ratings index

(5 - price)	Name
3	McGillin's
3	Di Nardo's Seafood
2	Alma de Cuba
2	Sotto Varalli
1	Moshulu

(b) Price index

Execution proceeds as follows. We read the top-scoring items in the *ratings* index, resulting in **Alma de Cuba**, and in the *price* index, resulting in **McGillin's**. Retrieving **Alma de Cuba** we compute a score of $4 \cdot 0.5 + 2 \cdot 0.5 = 3$. **McGillin's** receives a score of 3.5. We add these two results to a priority queue, prioritizing according to their scores. We set our threshold value τ to the cost function over the lowest scores read sequentially over the indices: $4 \cdot 0.5 + 3 \cdot 0.5 = 3.5$. Nothing exceeds our threshold, so we must read more values.

Suppose we next read **Moshulu** (score = 2.5) and **Di Nardo's Seafood** (score = 2.5). The threshold still has not changed at this point, so we cannot output any results, but we enqueue these (breaking ties arbitrarily). Next we read **Sotto Varalli** (score = 2.75) from both of the lists and add it to the priority queue, and the threshold gets lowered to 2.75. Now we have three items in the priority queue that all match or exceed 2.75: namely, **Alma de Cuba**, **McGillin's**, and **Sotto Varalli**. We output these and are done.

16.3 Keyword Search for Data Integration

Like many topics in data integration, the keyword search problem in this setting builds heavily on techniques applied to single databases. However, data integration exacerbates two challenges:

- In a single database, we can often rely on foreign keys to serve as the only edge types in the data graph. Across databases, we must typically *discover* edges through string matching techniques (Chapter 4) and other approaches resembling those of schema matching (Chapter 5). This may require additional processing to infer possible edges based on the query, and it results in a larger search space of potential query trees. Moreover, some of the inferred edges may in fact be incorrect.
- Across many databases, some sources are likely to represent different viewpoints and be of higher or lower relevance to a given query, based on the user's context and information need. Note that this is somewhat different from the general notion of authoritativeness or data quality, which can often be formulated in a query-independent way.

Hence the work on keyword-driven data integration typically considers how to tractably infer edges and how to use feedback on query results to correct edges or learn source relevance.

16.3.1 Scalable Automatic Edge Inference

Perhaps the most critical issue faced in providing keyword search in a data integration setting is that of discovering potential cross-source joins, such that user queries might integrate data from multiple sources. Unlike in a conventional data integration scenario, there is typically no mediated schema and no schema mappings — hence the search system may need to work directly on the contents of the data sources.

Naturally, keyword-search-based data integration systems typically include a preindexing and preprocessing step, whose tasks include indexing search terms, searching for possible semantically meaningful joins that also produce answers, and creating data structures that describe the potential edges in the data graph. There may also be capabilities for incrementally updating the data graph as new sources are discovered.

Conceptually, the task of finding semantically meaningful joins is very close to the problem of schema matching and, more specifically, finding an alignment between attributes in different schemas. The goal is to find attributes that match semantically and in terms of datatypes.

However, there are some key differences that in fact make the problem slightly simpler in this context. In both cases the goal is to find attributes that capture the same semantic concept. However, the join discovery problem is seeking key/foreign-key relationships between conceptually heterogeneous tuples to be joined, whereas the schema matching problem is one of trying to find a transformation that lets us convert tuples into a homogeneous form that can be unioned together. Moreover, in schema matching there is often a problem that the actual data values between a pair of source attributes may be non-overlapping (e.g., because the sources have complementary data that can be unioned together), whereas for join discovery we are only looking for situations in which there is overlap (hence joining on the attributes produces answers).

The task of discovering possible joins can be broken into two subtasks whose results are to be combined: discovering compatible data values and considering semantic compatibility.

DISCOVERING COMPATIBLE DATA VALUES

The first task seeks to identify which attributes come from the same general value domain, in effect clustering by the value domain. A challenge lies in how to do efficient comparisons of data values at scale. One approach is to compute statistical synopses of the values (using hashing, histograms, sketches, etc.) and another is to partition the data values across multiple machines and use parallel processing (map/reduce or alternatives) to compute the ratio of values that overlap.

CONSIDERING SEMANTIC COMPATIBILITY

Sometimes, attributes can have overlapping values without being semantically related (e.g., many attributes may be integer-valued). Here, we would like to consider other sources of evidence for compatibility, e.g., relatedness of attribute names. Such information can be obtained from schema mapping tools.

COMBINING EVIDENCE

The simplest means of combining data value compatability and semantic compatibility is as a weighted sum of the component scores. Alternatively, one can use link analysis techniques from recommendation systems, like label propagation. Specifically, we create a graph encoding of the data-to-data relationships and the metadata-to-metadata relationships and use the label propagation algorithm to combine the results and create a combined prediction of overlap.

16.3.2 Scalable Query Answering

Given a data graph with appropriate weights, there can often be a challenge in computing the top-scoring or top-k query answers: there may be a need to pose large numbers of queries simultaneously. The approaches described in Section 16.2 remain useful here, but the challenge is in how to only execute the work needed. If we have information about the values at the sources, we can often estimate a score range for certain queries or subqueries. Then during the generation and execution of top-k queries, we will only enumerate or execute the subqueries if they are able to contribute to the final answer.

16.3.3 Learning to Adjust Weights on Edges and Nodes

Earlier in this section, we described several means of precomputing weights for the edges and nodes in a data graph. One major issue is that it may be difficult to make correct assignments without involving data or domain experts and seeing how relevant a given source's data are with respect to a particular question. Recent work has shown that it is possible to make weight assignments *at query time* instead of purely during preprocessing. The idea is to provide "most likely" answers to user queries and request that the users (who may often be able to assess the plausibility of a query answer) provide feedback on any answers they know to be good or bad.

The system can then generalize from the feedback using machine learning techniques and will adjust certain "score components" that make up the weights on some of the edges in the data graph. More formally, the process is an instance of so-called *online* or *semi-supervised* learning (learning incrementally based on multiple feedback steps), and the "score components" are called features in the machine learning literature. A given edge in the data graph receives its weight as a particular combination of weighted features, e.g., the weight assigned to an edge between two metadata items may represent a weighted linear combination of (1) Jaccard distance between sets of values for these metadata items, (2) the string edit distance between the labels, or (3) the authoritativeness of the sources of the two metadata items and their values. Observe that some of these features might be specific to the edge between the nodes, and others might be shared with other edge weights (e.g., the source authoritativeness). The right set of features allows the system to generalize feedback on a single answer to other, related answers.

Two technical challenges must be solved in making this approach work.

It Must be Possible to go from Tuple Answers to the Features Forming the Tuple Score
Given a query result tuple, it is vital to be able to identify how and from where the tuple was produced and the individual features and weights that formed the score of the query result. The vector of features and weights, as well as feedback on the correct answers, will be given

to the online learning algorithm. Data provenance (Chapter 14) annotations on the output tuples can provide this capability of looking up feature values from query results.

The Scoring Model and Learning Algorithm Must be Closely Related
There are a wide variety of techniques used for online learning. However, in general it should be the case that the scoring model used to compute ranked results and the learning algorithm are compatible. When the scoring model is based on a weighted linear combination of components (the most popular model used in keyword search over databases), it is possible to use learning algorithms that rely on linear combinations of features, such as the Margin-Infused Ranking Algorithm (MIRA) to make weight adjustments. MIRA takes feedback in the form of constraints over output tuple scores ("the 5th-ranking answer should score higher than the 2nd-ranking answer") and adjusts feature weights to satisfy the constraint.

It has been shown that the learning techniques of this section can effectively address a variety of weight-adjustment problems for keyword-based data integration: learning node weights (e.g., source authoritativeness) across all queries, learning the relevance of nodes and edges to *specific* queries posed with a specific context, and learning the correct similarity score on an edge in the graph, e.g., to "repair" a bad value alignment.

Bibliographic Notes

Keyword search and its relationship to databases and data integration have been studied in a wide variety of ways. Yu, Lu, and Chang [587] provide a recent overview of work in the broader keyword search-over-databases field. Early systems included DISCOVER [304] and DbXplorer [18], which focused on SQL generation for keyword search but used very simple ranking schemes derived from the structure of the query tree itself. SPARK [403] proposed a scoring scheme that was motivated by techniques from information retrieval.

Algorithms for scaling up search over databases remain a key focus. BANKS [80] developed the techniques for backwards graph expansion described in this chapter and adopted a very general tree-based scoring model. A refined version of BANKS developed the more general bidirectional search strategy [337]. BLINKS [298] used the root-to-path-based scoring model described in this chapter and showed that there were complexity bound benefits. The STAR [344] Steiner Tree approximation algorithm exploits an existing taxonomic tree to provide improved performance beyond the methods described in this chapter and even gives an approximation guarantee in this setting. It focuses on supporting keyword search in the YAGO project [534], also discussed in Chapter 15. The work in [51] seeks to scale up keyword search in the DISCOVER system by requiring results to be returned within a time budget, and then to point the user toward a list of other resources for more specialized information, such as query forms matching the keywords [137, 492].

Other work has explored broader sets of operations in the keyword search context. SQAK [540] incorporates aggregation operations. The Précis system [523] developed a

semantics that enabled combinations of keywords using Boolean operators. Work on product search [509, 583] has explored taking keyword search terms and expanding them into selection predicates over multiple attributes, for a single table.

One of the major ingredients of keyword search systems is top-k query processing algorithms. Fagin's Threshold Algorithm and numerous variants appear in [221]. There are also a wide variety of approaches to performing joins in a ranked model, including [226, 266, 310, 415, 514, 515]. Since a top-k query should *only* execute to the point where it has generated k answers, specialized techniques [109, 311, 383] have been developed for query optimization in this setting and for estimating how much input actually gets consumed. Techniques have also been developed for storing indices over which top-k queries may be computed, while eliminating the need for random accesses [334]. The work [52] discusses how to build industrial-strength keyword search systems.

In the data integration setting, one task is to identify the domain of the query, and then separate its different components. For example, to answer the query "vietnam coffee production," we would first need to identify that the query is about the coffee domain, then translate it to a query of the form **CoffeeProduction(vietnam)**. We then need to find a table that is relevant to the property **CoffeeProduction** (which is likely to be called something very different in the table) and has a row for Vietnam. Initial work in this topic is described in [167]. The Kite system [512] developed techniques for using synopses to discover join relationships and to iteratively generate queries during top-k computation. The Q system [538, 539] developed the feedback-based learning approach described here. Work by Velegrakis et al. [69] considered the issue of how to perform ranked keyword search when only metadata are available for querying, as in certain data integration scenarios. Several works discussed in Chapter 15 addressed the problem of keyword queries on the collection of HTML tables found on the Web [101, 102, 391].

Naturally, there are connections between search, ranking, and probabilistic data (Chapter 13). Details on the use of probabilistic ranking schemes are provided in [312].