Proximity Search in Databases

Roy Goldman, Narayanan Shivakumar,
Suresh Venkatasubramanian, Hector Garcia-Molina

Stanford University
{royg, shiva, suresh, hector}@cs.stanford.edu

Abstract

An information retrieval (IR) engine can rank documents based on textual proximity of keywords within each document. In this paper we apply this notion to search across an entire database for objects that are "near" other relevant objects. Proximity search enables simple "focusing" queries based on general relationships among objects, helpful for interactive query sessions. We view the database as a graph, with data in vertices (objects) and relationships indicated by edges. Proximity is defined based on shortest paths between objects. We have implemented a prototype search engine that uses this model to enable keyword searches over databases, and we have found it very effective for quickly finding relevant information. Computing the distance between objects in a graph stored on disk can be very expensive. Hence, we show how to build compact indexes that allow us to quickly find the distance between objects at search time. Experiments show that our algorithms are efficient and scale well.

1 Introduction

Proximity search is successfully used in information retrieval (IR) systems to locate documents that have words occurring "near" each other [Sal89]. In this paper we apply this notion to search across an arbitrary database for objects that are "near" other objects of interest. Just as the distance between words in a document is an approximation of how related the terms are in the text, proximity search across an entire database gives a rough or "fuzzy" measure of how related objects are. While some situations demand "precise" query results, more and more online databases—such as content databases on the Web—enable users to interactively browse results and submit refining queries. In these settings, proximity estimates can be very useful for *focusing* a search. For example, we may be looking for a "person" with a last name that sounds like "Schwartz" but may not know if this person is an employee, a manager, or a customer. A search may yield many people, spread out throughout the database. If we also know that the target person is somehow related, say, to a particular complaint record, then we can narrow down the original set, ranking it by how closely related each person is to the complaint. Similarly, in a database that tracks overnight package delivery, we may wish to locate any information pertinent to a lost package (e.g., people that handled it, locations it went through, customers that signed for it) ranked by how relevant the information is to the lost package.

For object-proximity searching, we view the database simply as a collection of objects that are related by a *distance* function. The objects may be tuples, records, or actual objects, or even fields within these structures, if finer granularity is desired. The distance function is provided by the system or an administrator; it indicates how "closely related" certain (not necessarily all) pairs of objects are. For instance, in a personnel database, the number of links that separate objects may be a good measure of how closely they are related. Two employees working in the same department are closely related (each employee is linked to the same department); if two departments cooperate on the same product, then an employee in one department is related to an employee in the other, but to a lesser extent. We can also weight each type of link to reflect its semantic importance. In a relational context, tuples related by primary-key/foreign-key dependencies could be considered closely linked, while tuples in the same relation could also be related, to a lesser extent.

Traditional IR proximity search is intra-object, i.e.,

**Proceedings of the 24th VLDB Conference
New York, USA, 1998**

it only considers word distances within a document. Our search is inter-object, i.e., we rank objects based on their distance to other objects. This difference introduces two related challenges, which are the main focus of this paper.

- *Distance Computation*: Text intra-object distance is measured on a single dimension. Thus, it is easy to compute distances between words if we simply record the position of each word along this one dimension. For inter-object search, we measure distance as the length of the shortest path between objects.
- *Scale of Problem*: For efficient inter-object proximity search, we need to build an index that gives us the distance between *any* pair of database objects. Since there can be a huge number of objects, computing this index can be very time consuming. For intra-object search, on the other hand, we only need to know the distance between words within an object, a much smaller problem.

In this paper we describe optimizations and compression schemes that allow us to build indexes that can efficiently report distances between any pair of objects. Experiments show that our algorithms have modest time and space requirements and scale well.

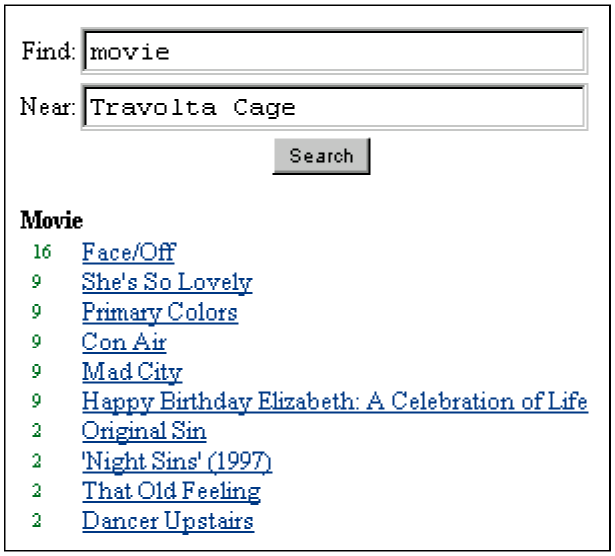
In Section 2, we trace an example over a sample database, to further motivate inter-object proximity search. Section 3 then defines our problem and framework in more detail. In Section 4, we illustrate a particular instance of our general framework, as applied to keyword searching over databases. Section 5 details our algorithms for efficient computation of distances between objects, and experimental results are given in Section 6. We discuss related work in Section 7.

2 Motivating Example

The Internet Movie Database (www.imdb.com) is a popular Web site with information about over 140,000 movies and over 500,000 film industry workers. We can view the database as a set of linked objects, where the objects represent movies, actors, directors, and so on. In this application it is very natural to define a distance function based on the links separating objects. For example, since John Travolta stars in the movie "Primary Colors," there is a close relationship between the actor and the movie; if he had directed the movie, the bond might be tighter.

Within our framework, proximity searches are specified by a pair of queries:

- A *Find query* specifies a *Find set* of objects that are potentially of interest. For our example, let us say that the find query is keyword-based. For instance, "*Find* movie" locates all objects of type "movie" or objects with the word "movie" in their body.


Find: movie

Near: Travolta Cage

Search

Movie

16	Face/Off
9	She's So Lovely
9	Primary Colors
9	Con Air
9	Mad City
9	Happy Birthday Elizabeth: A Celebration of Life
2	Original Sin
2	'Night Sins' (1997)
2	That Old Feeling
2	Dancer Upstairs

Figure 1: Results of proximity search over the Internet Movie Database

- Similarly, a *Near query* specifies a *Near set.* The objective is to rank objects in the *Find* set according to their distance to the *Near* objects. For our examples we assume the near query is also keyword-based.

For example, suppose a user is interested in all movies involving both John Travolta and Nicolas Cage. This could be expressed as "*Find* movie *Near* Travolta Cage." Notice that this query does not search for a single "movie" object containing the "Travolta" and "Cage" strings. In this database, the person named "Travolta" is represented by a separate object. Similarly for "Cage." Movie objects simply contain links to other objects that define the title, actors, date, etc. Thus, the proximity search looks for "movie" objects that are somehow associated to "Travolta" and/or "Cage" objects.

To illustrate the effect of this query, it is worthwhile to jump ahead a bit and show the results on our implemented prototype. The details of this system are described in Section 4; the database contains the IMDB subset referring to 1997 films. Figure 1 shows the query "*Find* movie *Near* Travolta Cage" along with the top 10 results. As we might expect, "Face/Off" scored highest since it stars both actors. That is, both actor objects are a short distance away from the "Face/Off" movie object. The next five movies all received the same second-place score, since each film stars only one of the actors. (See Section 3 for a detailed explanation of how ranking works.) The remaining movies reflect indirect affiliations—that is, larger distances. "Original Sin," for example, stars Gina Gershon, who also played a part in "Face/Off." In Section 4 we give addi-

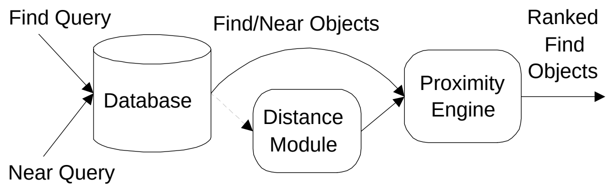


Figure 2: Proximity search architecture

tional examples of queries and results, over a different database.

Proximity searches are inherently fuzzy. If one can precisely describe the desired information (e.g., what relation it occurs in, the exact path to it, the precise contents of fields) then traditional database queries will usually be best. Still, proximity search is very useful when it is impractical to generate a specific query, or when a user simply wants to search based on the general relevance of different data objects.

Current database and IR systems do not provide inter-object proximity search. Often, applications implement particular versions of proximity search. For example, the IMDB Web site does offer a form for searching for movies with multiple specified actors. Our goal is to provide a general-purpose proximity service that could be implemented on top of any type of database system.

3 The Problem

The basic problem is to rank the objects in one given set (the *Find* set) based on their proximity to objects in another given set (the *Near* set), assuming objects are connected by given numerical "distances." We first discuss our conceptual model in detail, and then we formalize our notion of proximity.

3.1 Conceptual Model

Figure 2 shows the components of our model. An existing database system stores a set of objects. Applications generate *Find* and *Near* queries at the underlying database. (In our motivating example, these queries were keyword searches.) The database evaluates the queries and passes *Find* and *Near* object result sets (which may be ranked) to the proximity engine. Database objects are opaque to the proximity engine, which only deals with object identifiers (OIDs).[1] The proximity engine then re-ranks the *Find* set, using distance information (and possibly the initial ranks of the *Find* and *Near* objects). The distance information is provided by a distance module. Conceptually, it provides the proximity engine a set of triplets (X, Y, d), where d is the known distance between database objects with identifiers X and Y. (Note that the distance module uses the same identifiers as the database system.) We assume that all given distances are greater than or equal to 1. The proximity engine then uses these base distances to compute the lengths of shortest paths between all objects. Because we are concerned with "close" objects, we assume the distance between any two objects to be exact only up to some constant K, returning ∞ for all distances greater than K. This assumption enables improved algorithms, as described in Section 5.

[1] Most relational systems do not expose explicit row identifiers; we can use primary key values or "signatures," e.g., checksums computed over all tuple field values. Individual fields can be identified simply by their values.

To the proximity engine, the database is simply an undirected graph with weighted edges. This does not mean that the underlying database system must manage its data as a graph. For example, the database system may be relational, as illustrated by the left side of Figure 3. This shows a small fragment of a normalized relational schema for the Internet Movie Database. The right side of the figure shows how that relational data might be interpreted as a graph by the search engine. Each entity tuple is broken into multiple objects: one entity object and additional objects for each attribute value. Distances between objects are assigned to reflect their semantic closeness. For instance, in Figure 3 we assign small weights (indicating a close relationship) to edges between an entity and its attributes, larger weights to edges linking tuples related through primary and foreign keys, and the largest weights to edges linking entity tuples in the same relation. (For clarity, the graph shows directed, labeled edges; our algorithms ignore the labels and edge directions.) Of course, the distance assignments must be made with a good understanding of the database semantics and the intended types of queries. It is simple to model object-oriented, network, or hierarchical data in a similar manner.

3.2 Proximity and Scoring Functions

Recall that our goal is to rank each object f in a *Find* set F based on its proximity to objects in a *Near* set N. Each of these sets may be ranked by the underlying database system. We use functions r_F and r_N to represent the ranking in each respective set. We assume these functions return values in the range $[0, 1]$, with 1 representing the highest possible rank. We define the distance between any two objects $f \in F$ and $n \in N$ as the weight of the shortest path between them in the underlying database graph, referred to as $d(f, n)$. To incorporate the initial rankings as well, we define the *bond* between f and n ($f \neq n$):

$$b(f,n) = \frac{r_F(f) r_N(n)}{d(f,n)^t} \qquad (1)$$

(We set $b(f, n) = r_F(f) r_N(n)$ when $f = n$.) A bond ranges from $[0, 1]$, where a higher number indicates

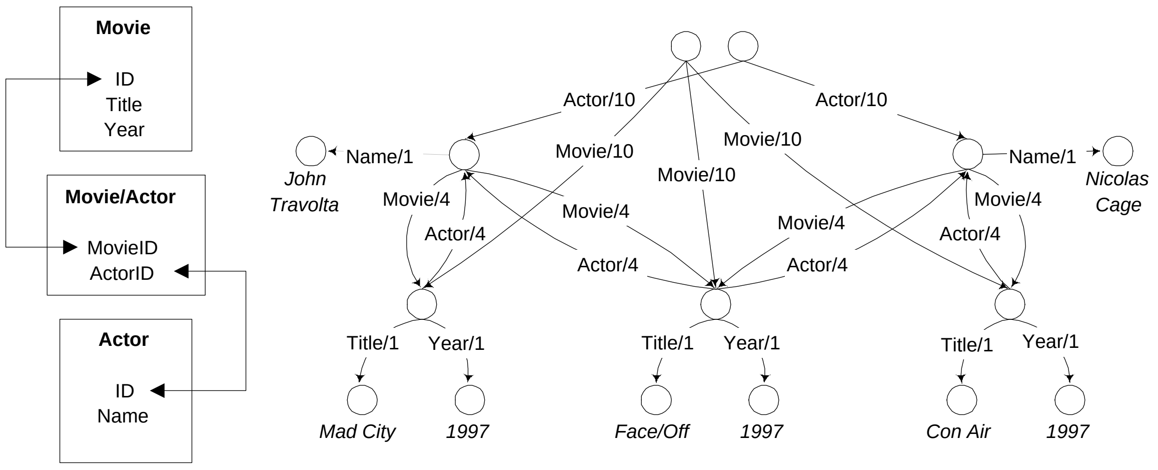


Figure 3: A fragment of the movie database relational schema and a database instance as a graph

a stronger bond. The tuning exponent t is a nonnegative real that controls the impact of distance on the bond.

While a bond reflects the relationship between two objects, in general we wish to measure proximity by scoring each *Find* object based on all objects in the *Near* set. Depending on the application, we may wish to take different approaches for interpreting bonds to the *Near* objects. We discuss three possible scoring functions:

- *Additive*: In the query from our motivating example to "*Find* movie *Near* Travolta Cage," (Section 2), our intuition leads us to expect that a film closely related to both actors should score higher than a film closely related to only one. To capture this intuition, we score each object f based on the sum of its bonds with *Near* objects:

$$score(f) = \sum_{n \in N} b(f,n) \tag{2}$$

  Here the score can be greater than 1.

- *Maximum*: In some settings, the maximum bond may be more important than the total number. Thus, we may define

$$score(f) = \max_{n \in N} b(f,n) \tag{3}$$

  In this case, scores are always between 0 and 1.

- *Beliefs*: We can treat bonds as beliefs [Goo61] that objects are related. For example, suppose that our graph represents the physical connections between electronic devices, such that two objects close together in the graph are close together physically as well. Assume further that r_N gives our belief that a *Near* device is faulty (1 means we are sure it is faulty). Similarly, r_F can indicate the known status of the *Find* devices. Then, for a device $f \in F$ and a device $n \in N$, $b(f,n)$ may give us the belief that f is faulty due to n, since the closer f is to a faulty device, the more likely it is to be faulty. Our belief that f is faulty (between 0 and 1), given the evidence of all the *Near* objects, is:

$$score(f) = 1 - \prod_{n \in N} (1 - b(f,n)) \tag{4}$$

Of course other scoring functions may also be useful, depending on the application. We expect that the proximity search engine will provide several "standard" scoring functions, and that users submitting queries will specify their intended scoring semantics. This is analogous to how users specify what standard function (e.g., COUNT, MAX, AVG) to use in a statistical query.

4 Keyword Search Application

This section describes a prototype that implements our framework, as first mentioned in Section 2. By connecting to our system on the Web, users can search databases by specifying *Find* and *Near* keywords. Those keywords are used to generate corresponding input object sets for our proximity engine, which then ranks *Find* objects by their relevance to the *Near* objects.

We implemented our proximity architecture on top of *Lore* [MAG+97], a database system designed at Stanford for storage and queries of graph-structured data. Lore's data model is the *Object Exchange Model (OEM)* [PGMW95], originally designed at Stanford to facilitate integration of data from heterogeneous sources. An OEM database is essentially a directed graph, with data objects linked by textually labeled edges that describe relationships. In OEM, atomic

Find picture Near China	Photos of 6 Chinese students, followed by Prof. Widom, who advises 3 of them, and Prof. Ullman, who advises 2
Find publication Near Garcia	All of Prof. Garcia-Molina's publications, followed by publications of his students
Find publication Near Garcia Widom	The top publications are co-authored by Profs. Garcia-Molina and Widom, followed by their individual papers
Find group_member Near September	The top results are members born in September
Find publication Near OEM	The top pub. has "OEM" in its title, followed by a pub. stored in "oem.ps," followed by one with keyword "oem"

Figure 4: Stanford Database Group keyword searches

data such as integers, reals, strings, or images are stored only in leaf objects. An OEM database isn't forced to conform to any prespecified schema; hence, it is useful for *semistructured* data, which may have some structure but may also contain irregularities. The graph from Figure 3 is in fact an OEM database, though we have augmented the model to support weights on edges.

To generate the *Find* and *Near* sets for our proximity measurement, our application simply takes keywords as input. Note that in an OEM database, a keyword could identify an object with a specific incoming edge label, an atomic object whose data contains the keyword, or both. For each keyword, we use Lore indexes to add to the *Find* or *Near* set all objects with a matching incoming label and all atomic objects containing the specified keyword. Currently, Lore does not rank the objects returned by a keyword lookup; hence we assign all objects an initial rank of 1.

Based on informal usability tests, we chose to set t to 2 in our bond definition (Equation 1), to weight nearby objects more heavily; this setting causes a bond to drop quadratically as distance increases. To capture the intuition given in the motivating example, we use the additive scoring function (Equation 2) to score each *Find* object. Together, our choice of tuning parameter and scoring function will give a *Find* object f_1 that is 1 unit away from a *Near* object twice the score of an object f_2 2 units away from two objects. In the user interface, we linearly scale and round all scores to be integers.

Figure 4 summarizes the results of several keyword search queries over a database describing the members, projects, and publications of the Stanford Database Group (DBGroup). The database has been built from scratch in OEM, containing about 4200 objects and 3600 edges. Initial supplied distances are similar to those shown in Figure 3. Examples show that proximity search is a useful complement to traditional database queries, allowing users to narrow in on relevant data without having to understand the nature of all database relationships, and without fully specifying structural queries. In this interactive setting, users can easily browse results and submit additional queries. Note that this application reflects just one particular set of choices for instantiating our proximity model—how we generate the *Find*/*Near* sets, our initial ranking functions r_F and r_N, our tuning exponent t in the bond definition, and our choice of scoring function. Our keyword search application is available to the public at http://www-db.stanford.edu/lore; users can submit their own searches and browse the results.

5 Computing Object Distances

For our proximity computations to be practical, we need to efficiently find the distances between pairs of objects. In this section we discuss the limitations of naive strategies and then focus on our techniques for generating indexes that provide fast access at search time.

First, we discuss the framework for our distance computations. As described in Section 3.1, the proximity engine takes as input *Find* and *Near* sets of OIDs, and a set of base distances between objects. Let V be the set of objects. We assume the distances are provided by the distance module of Figure 2 as an *edge-list* relation E_1, with tuples of the form $\langle u, v, w\rangle$, if vertices $u, v \in V$ share an edge of weight w. For convenience, we assume that E_1 contains $\langle u, v, w\rangle$, if $\langle v, u, w\rangle$ is in E_1. Let G refer to the graph represented by E_1.

In graph G, we define $d_G(u, v)$ to be the shortest distance between u and v. (We will drop the subscript G if it is clear which graph we are referring to.) As mentioned in Section 3.1, our proximity search focuses on objects that are "close" to each other. Hence, we assume all distances larger than some K are treated as ∞. In our prototype, setting $K = 12$ for the IMDB and DBGroup databases yields reasonable results, given the initial supplied distances.

5.1 Naive Approaches

At one extreme, we could answer a distance query by performing all required computation at search time. A classical algorithm to compute the shortest distance between two vertices is Dijkstra's single-source shortest path algorithm [Dij59]. While the algorithm is efficient for graphs in main memory, computing the shortest distance for an arbitrary disk-based graph could take as many as $|E_1|$ random seeks. There have been recent attempts to reduce I/O in the disk-based version of the algorithm using *tournament trees* [KS]; however, these attempts still require many random seeks.

```
Algorithm: Distance self-join
Input: Edge set $E_1$, Maximum required distance: $K$
Output: Lookup table $Dist$ supplies the shortest distance (up to $K$) between any pair of objects
[1] For $l = 1$ to $\lceil \log_2 K \rceil$
[2]     Copy $E_l$ into $E'_{l+1}$.
[3]     Sort $E_l$ on first vertex. // To improve performance
[4]     Scan sorted $E_l$:
[5]         For each $\langle v_i, v_j, w_k \rangle$ and $\langle v_i, v'_j, w'_k \rangle$ in $E_l$ where $v_j \neq v'_j$
[6]             If $(w_k + w'_k \leq 2^l)$ and $(w_k + w'_k \leq K)$
[7]                 Add $\langle v_j, v'_j, w_k + w'_k \rangle$ and $\langle v'_j, v_j, w_k + w'_k \rangle$ to $E'_{l+1}$.
[8]     Sort $E'_{l+1}$ on first vertex, and store in $E_{l+1}$.
[9]     Scan sorted $E_{l+1}$:
[10]        Remove tuple $\langle u, v, w \rangle$, if there exists another tuple $\langle u, v, w' \rangle$, with $w > w'$.
[11] Let $Dist$ be the final $E_{l+1}$.
[12] Build index on first vertex in $Dist$.
```

Figure 5: "Self-Join" distance precomputation

A better approach would be to precompute shortest distances between all pairs of vertices and store them in a lookup table for fast access. The classical algorithm to compute all-pairs shortest distances is Floyd-Warshall's dynamic programming based algorithm [Flo62]. An obvious disk-based extension of the algorithm requires $|V|$ scans of G. Clearly this is inefficient, and there is no simple way to modify the algorithm to find only distances no larger than K. There has been much work on the related problem of computing the transitive closure of a graph. In Section 7 we discuss these approaches and why they are not suitable for our problem.

In the next section, we propose an approach for precomputing all-pairs distances of at most K that is efficient for disk-based graphs, using well-known techniques for processing "self-joins" in relational databases. Section 5.3 shows how we can exploit available main memory to further improve both the space and time requirements of index construction.

5.2 Precomputing Distances Using "Self-Joins"

We use the following idea as the basic step for precomputing all-pairs shortest distances. We will assume that K is a power of two for ease of exposition; of course, our algorithms work for general K as well. Let A be the adjacency matrix of G; for any $v_i, v_j \in V$, $A[v_i][v_j] = w$ if an edge $\langle v_i, v_j, w \rangle$ exists. Else, if $i = j$, $A[v_i][v_j] = 0$, else $A[v_i][v_j] = \infty$. Given A, we compute A^2, where the matrix multiplication is taken over the closed semiring of $R^+ \cup \{\infty\}$, with scalar addition and multiplication replaced by the min operator and scalar addition respectively [AHU74]. Observe that for any pair (v_i, v_j) in G, A^2 contains the shortest distance between v_i and v_j that goes through at most one other vertex. Similarly, we can generate A^4 by squaring A^2, and so on, until we obtain A^K.

Figure 5 presents our implementation of the above idea, using simple self-join techniques. Roughly, Steps [2] – [10] correspond to the basic matrix multiplication idea we just described. E_l corresponds to the edge-list representation of $A^{2^{l-1}}$, and E'_l corresponds to the edge-list representation of $A^{2^{l-1}}$ before applying the min operator. (We will soon see what they mean intuitively.) In Steps [5] – [7], we are generating tuple $\langle v_j, v'_j, w_k + w'_k \rangle$, since we know that the shortest distance between v_j and v'_j cannot exceed $w_k + w'_k$ (due to a path through v_i). Step [6] restricts our selection to weights in the desired range. In Steps [8] – [10], we eliminate non-shortest distances between vertex pairs. By iterating the above steps $\lceil \log_2 K \rceil$ times (Step [1]), we square the original A matrix $\lceil \log_2 K \rceil$ times, obtaining A^K. Because all initial distances are at least 1, the final matrix is guaranteed to contain all shortest distances at most K. The final output *Dist* of the above algorithm is a distance lookup table that stores the *K-neighborhoods* of all vertices. That is, the table stores all $\langle v_i, v_j, w_k \rangle$ for all vertex pairs v_i, v_j with shortest path length w_k units ($w_k \leq K$). For convenience, we will sometimes refer to E'_l as the *unzapped* edge-list, and we refer to E_l as the corresponding *zapped* edge-list, with non-shortest distances removed.

The above procedure runs with little I/O overhead, since sorting the data enables sequential rather than random accesses. Note that other efficient techniques are possible for computing the self-join (such as hash joins), and in fact given E_l we can use standard SQL to generate E_{l+1} [GSVGM98]. Querying for $d(v_i, v_j)$ is also efficient—since we index the *Dist* table, we can access the neighborhood of v_i, and look for a tuple of the form $\langle v_i, v_j, w_k \rangle$. If there is such a tuple, we know the distance to be w_k. If no such tuple exists, the distance is greater than K, and we return ∞.

However, the construction of *Dist* could be expensive using the above approach, since in Step [5] – [7], we produce the cross-product of each vertex neighborhood

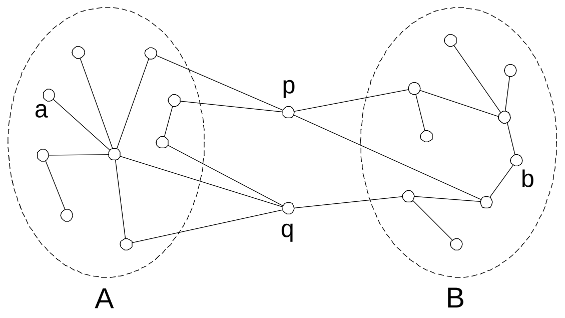


Figure 6: Hub vertices

with itself. The size of such a cross-product could be as large as $|V|^2$ in the worst-case. For instance, when we executed the self-join algorithm on the the 4MB edge-list for the IMDB database described in Section 2 for $K = 8$, the edge-list grew to about one gigabyte—250 times larger than the initial input! In the next section, we propose a technique to alleviate this problem.

5.3 Hub Indexing

We now propose *hub indexing*, which allows us to encode shortest distances in far less space than required by the self-join algorithm, with little sacrifice in access time. We use Figure 6 to explain what *hubs* are and how they can be used to compute distances efficiently. If we execute our simple self-join algorithm from the previous section on the given graph, we will explicitly store the $|A| \times |B|$ pair-wise shortest distances from vertices in A to those in B. (We also store distances for pairs of objects both in A or both in B.) Computing $d(a, b)$ for some $a \in A$ and $b \in B$ merely involves checking the *Dist* table for a tuple of the form $\langle a, b, w\rangle$, as described earlier.

In Figure 6 we see that if we remove p and q, the graph is disconnected into two sub-graphs A and B. Rather than storing all $|A| \times |B|$ distances, suppose we store only the $|A| + |B|$ shortest distances to p, the $|A| + |B|$ shortest distances to q, and the shortest distance between p and q. Note that space savings are maximized when $|A| = |B|$. Of course, the query procedure for such an approach is slightly more complex. We can see that the shortest-path between a and b can be one of $a \sim p \sim b$ (not through q), $a \sim q \sim b$ (not through p), $a \sim p \sim q \sim b$, or $a \sim q \sim p \sim b$. We can compute $d(a, b)$ by finding these four distances and choosing the smallest.

The above description gives the reader a rough idea of our approach. By finding hubs such as p and q, we can sharply reduce the storage required for a distance index, and we will show how to efficiently handle the more complex query procedure. In addition, we can store hubs and the shortest distances between them in main memory. As we allocate more memory for hub storage, our index shrinks and query times decrease as well. Effectively choosing hubs in an arbitrary graph is a challenging problem, an issue we defer to Section 5.3.4. Assuming we have a set of hubs, the following sections describe how to build a hub index and then answer distance queries using it.

5.3.1 Constructing Hub Indexes

As suggested by the above discussion, a hub index is comprised of two key components: a hub set H (and the shortest distance between each pair of its elements) and a table of distances between pairs of objects whose shortest paths do not cross through elements of H. For simplicity, we redefine the *Dist* lookup table from Section 5.2 to be this new table. The correctness of our hub index creation algorithm (and the corresponding query procedure given in the next section) is proven in [GSVGM98].

Given H, we can reuse the algorithm of Figure 5 almost verbatim to construct the new *Dist* table. The only required change is to Step [6], which we replace with

$$[6'] \quad \text{If } (w_k + w'_k \leq 2^l) \text{ and } (w_k + w'_k \leq K) \text{ and } v_i \notin H$$

By checking that v_i is not in H we make sure that we do not consider any paths that cross hubs. (Paths with hubs as endpoints are still considered.) For each $v \in V$, *Dist* stores all vertices reachable within a distance of K without crossing any hubs; we call this set of vertices the "hub-bordered" neighborhood of v.

As we will explain in the next section, pair-wise distances between hubs must be consulted many times to evaluate a distance query. Fortunately, experiments discussed in Section 6 show that even a small set of hubs greatly reduces index size. Hence, our query algorithm assumes that the pair-wise distances of all hubs are available in main memory. We wish to build a square adjacency matrix *Hubs* such that $Hubs[h_i][h_j]$ gives the shortest distance between hubs h_i and h_j. To do so, we first initialize each entry of *Hubs* to ∞. Then, with one sequential scan of *Dist*, for each edge $\langle h_i, h_j, w_k\rangle$, where $h_i, h_j \in H$, we set $Hubs[h_i][h_j] = w_k$. This step "short-cuts" the need to recompute all distances from scratch. Finally, we use Floyd-Warshall's algorithm to compute all-pairs shortest distances in *Hubs*. Floyd-Warshall works in-place, without requiring additional memory. Since H is typically small and engine initialization occurs rarely, we are generally not concerned with the time spent computing *Hubs* from H and *Dist*. Still, we have the option of fully materializing *Hubs* at index creation time and then loading it directly into memory at engine initialization.

Since we keep hubs and their distances in memory, a hub index has the nice property that answering a distance query requires less work on disk as more memory is made available. In fact, if the entire adja-

```
Algorithm: Pair-wise distance querying
Input: Lookup table on disk: $Dist$, Lookup matrix in memory: $Hubs$,
       Maximum required distance: $K$, Hub set: $H$
       Vertices to compute distance between: $u, v$ $(u \neq v)$
Return Value: Distance between $u$ and $v$: $d$
[1] If $u, v \in H$, return $d = Hubs[u][v]$.
[2] $d = \infty$
[3] If $u \in H$
[4]     For each $\langle v, v_i, w_k \rangle$ in $Dist$
[5]         If $v_i \in H$     // Path $u \sim v_i \sim v$
[6]             $d = \min(d, w_k + Hubs[v_i][u])$
[7]     If $d > K$, return $d = \infty$, else return $d$.
[8] Steps [4] – [7] are symmetric steps if $v \in H$, and $u \notin H$.
[9] // Neither $u$ nor $v$ is in $H$
[10] Cache in main-memory $(E_u)$ all $\langle u, v_i, w_k \rangle$ from $Dist$
[11] For each $\langle v, v_i', w_k' \rangle$ in $Dist$
[12]     If $(v_i' = u)$
[13]         $d = \min(d, w_k')$     // Path $u \sim v$ without crossing hubs
[14]     For each edge $\langle u, v_i, w_k \rangle$ in $E_u$
[15]         If $v_i' \in H$ and $v_i \in H$     // Path $u \sim v_i \sim v_i' \sim v$ through hub vertices
[16]             $d = \min(d, w_k + w_k' + Hubs[v_i'][v_i])$
[17] If $d > K$, return $d = \infty$, else return $d$.
```

Figure 7: Pair-wise distance querying

cency matrix fits in memory, we can choose H to be V and eliminate query-time disk access entirely. Our approach reveals a smooth transition to Floyd-Warshall's algorithm as main memory increases. Engine administrators can specify a limit for the number of hub points based on available memory.

5.3.2 Querying Hub Indexes

Given the disk-based *Dist* table and the in-memory matrix *Hubs*, we can compute the distance between any two objects u and v using the algorithm in Figure 7. The algorithm performs a case-by-case analysis when it answers such queries. To help explain the algorithm, we refer back to the graph in Figure 6, assuming $H = \{p, q\}$. Steps [1] through [8] are straightforward, since these steps handle the case where one or both of u and v are in H. (In terms of Figure 6, suppose that u and/or v are in $\{p, q\}$.) Steps [10] through [17] address the case where neither input vertex is in H. Steps [12] – [13] consider the case where the shortest path from u to v does not go through any of the vertices in H and its distance is therefore explicitly stored in *Dist*. (In Figure 6, consider the case where both vertices are in A.) Steps [14] – [16] handle shortest paths through vertices in H, such as a path from any $a \in A$ to any $b \in B$ in the figure.

If both u and v are in H, no disk I/O is performed. Recall that *Dist* is indexed based on the first vertex of each edge. Hence, in case either u or v is in H, one random disk seek is performed to access the hub-bordered neighborhood of v or u (Steps [4] – [8]). In case neither is in H, two random disk seeks are performed to access the hub-bordered neighborhoods of both u and v (Steps [10] and Step [11]). The algorithm implicitly assumes that the hub-bordered neighborhood for any given vertex can be cached into memory (Step [10]). Since we use hubs, and given that K is generally small, we expect this assumption to be safe. Additional buffering techniques can be employed if needed.

5.3.3 Generalizing to Set Queries

While the previous section discusses how to use a hub index to look up the distance between a single pair of objects, a *Find/Near* query checks the distance between each *Find* and each *Near* object. For instance, we may need to look up the pair-wise distances between $Find = \{v_1, v_2\}$ and $Near = \{v_3, v_4, v_5\}$ The naive approach to answering such a query is to check the hub index for each of $\{v_1, v_3\}$, $\{v_1, v_4\}$, $\{v_1, v_5\}$, and so on. When we have F *Find* objects and N *Near* objects, this approach will require about $2 \times F \times N$ disk seeks, impractical if F and N are large. If the *Dist* table data for all of either the *Find* or the *Near* objects fits in main memory, we can easily perform all *Find/Near* distance lookups in $F + N$ seeks. If not, we can still buffer large portions of data in memory to improve performance.

In some cases, even $F + N$ seeks may still be too slow. Our movie database, for example, contains about 6500 actors. Hence, finding the result to a query like "*Find* actor *Near* Travolta" will take at least 6500 seeks. To avoid such cases, we allow engine administrators to specify object-clustering rules. For example, by clustering all "actors" together in *Dist* we can avoid random seeks and execute the queries efficiently. Our engine is general enough to cluster data arbitrarily based on user specifications. In our keyword proximity search application (Section 4), we cluster based on labels, such as "Actor," "Movie," "Producer," etc. Note that this approach increases the space requirements of *Dist*, because these clusters need not be disjoint. To mitigate the replication, preliminary investigation suggests that we can significantly compress vertex neighborhoods on disk, discussed further in Section 6.

5.3.4 Selecting Hubs

Recall that we allocate an in-memory matrix of size M for storage of hubs. Hence, for any graph, we can select up to $\sqrt{M}$ hubs. In this section, we summarize our strategy for hub selection.

Consider again the example of Figure 6. Suppose we had a procedure that could pick p and q as vertices that disconnect the graph into two "balanced" sub-graphs A and B. Given such a procedure, we could recursively disconnect both A and B in a similar manner to gain further savings. This recursion would generate a hierarchy of vertex sets, each of which disconnects a graph into two sub-graphs.

The set $\{p, q\}$ is known as a *separator* in graph theory. Formal definitions have been developed to characterize hierarchies of *balanced separators*, which guarantee that the disconnected subgraphs are similar in size. (See [GSVGM98] for more detail.) There exist linear time algorithms that compute balanced separators for graphs of constant treewidth [Bod96], and for planar graphs [LT80]. It can be shown that a balanced separator yields an optimal graph decomposition for in-memory distance queries [HKRS94, CZ95, Pel97]. Hence, balanced separators would be ideal candidates for hubs. For tree-shaped data, such as HTML or XML [Con97] documents, we can use the aforementioned tree-based separator algorithm to generate hubs.

Unfortunately, for arbitrary graphs, a nontrivial balanced separator theorem does not hold. The best known approximation yields a separator that is a factor of $O(\log n)$ larger than the minimum [AKR93]. Hence, we have designed an heuristic for selecting hubs that is efficient to implement and performs well in practice. The heuristic is to select up to $\sqrt{M}$ vertices with highest degree as hubs. We can make this selection with one scan of the edge-list. Our strategy serves two purposes. Firstly, notice that Steps [5] – [7] of the original self-join algorithm (Figure 5) generate $deg^2(v_i)$ tuples, where $deg(v_i)$ is the degree of vertex v_i. In the revised hub version of the algorithm, we avoid generating $deg^2(v)$ tuples for vertices of highest degree. Secondly, it is quite likely that high degree vertices lie on many shortest paths. Just like airline hub cities in a route map, vertices that lie on many shortest paths often effectively divide the graph into "balanced" subsets. Note that the correctness of our indexing algorithm does not depend on hubs actually separating a graph (see [GSVGM98] for proof); any vertex can in principle be chosen as a hub. Experiments for hub index creation are discussed in the next section. The results show that our hub selection heuristic is effective at reducing the time and space required to build an index.

6 Performance Experiments

We now study some performance related aspects of building hub indexes. Questions we address in this section include (1) Given a small, fixed number of hubs, what are the space and time requirements of index construction? (2) How do the algorithms scale with larger datasets? (3) What is the impact of selecting fewer or more hubs on the index construction time? (4) How fast is query execution? For our experiments, we used a Sun SPARC/Ultra II (2×200 MHz) running SunOS 5.6, with 256 MBs of RAM, and 18 GBs of local disk space.

We use the IMDB dataset to illustrate some of the trade-offs in this paper. We also experimented with the DBGroup dataset, but due to lack of space we do not present these results—however, the results were similar to those of IMDB. Since the IMDB dataset is small (its edge-list is about 4MB), we built a generator that takes as input IMDB's edge-list and scales the database by any given factor S. Note that we do not blindly copy the database to scale it; rather we create a "forest" by computing statistics on the small dataset and producing a new, larger dataset with similar characteristics. For instance, the percentage of popular actors will be maintained in the scaled-up version as well, and this set of actors will be acting in a scaled-up number of new movies. Similarly, movies will have the same distribution of actors from the common pool of S times as many actors; the ratio of "romance" movies to "action" movies will stay about the same. Since our generator produces the above graphs based on a real dataset, we believe it gives us a good testbed to empirically evaluate our algorithms. While we think the structure of our data is typical of many databases, of course it does not reflect every possible input graph.

First, we discuss index performance when the number of hubs is fixed at a "small" number. Recall from Section 5.3 that the algorithm requires temporary storage (for the unzapped edge-lists) before creating and indexing the final zapped edge-list. For our experiments, we build an ISAM index over the final edge-list. Figure 8 shows the temporary and final space requirements of a hub index for different values of K. We define the space required as a multiple of the size of the original input. For this graph, we set $S = 10$ and we choose no more than 2.5% of the vertices as hubs. For this case (about 40MB of data), we required less than 250K of main memory to store our *Hubs* matrix. We see that both the temporary and fi-

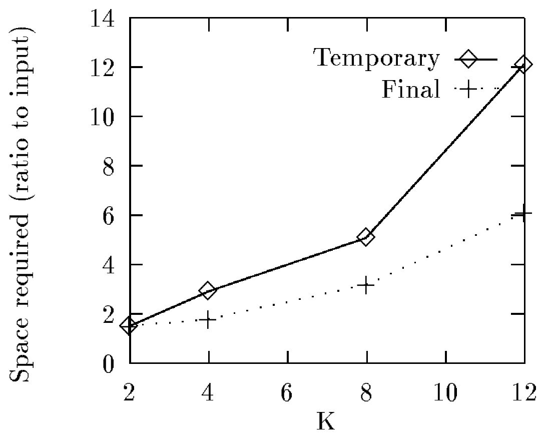


Figure 8: Storage requirements with varying K

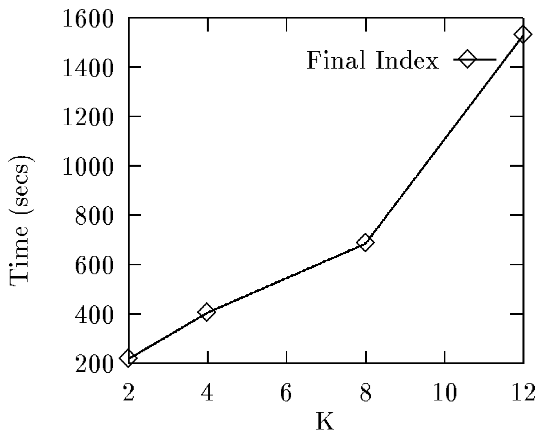


Figure 9: Index construction time with varying K

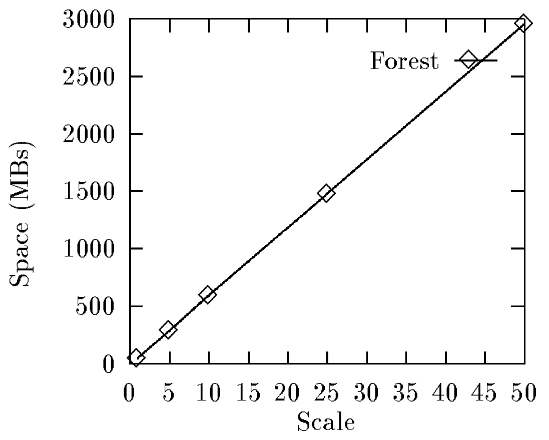


Figure 10: Total storage with varying scale

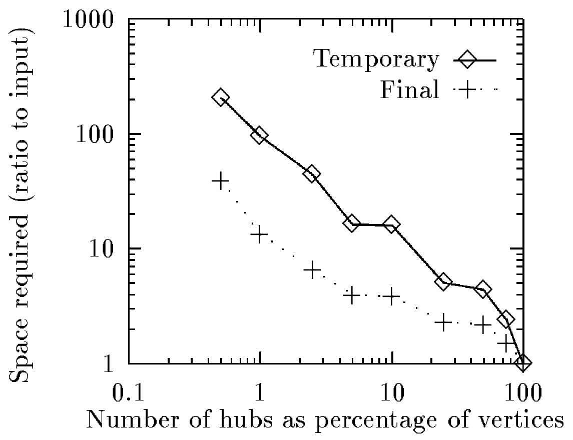


Figure 11: Space ratio with varying number of hubs

nal space requirements can get large. For $K = 12$ (the K used for our prototype in Section 4), the temporary and final space requirements are about 12 times and 6 times larger than the input edge-list, respectively. Similarly, Figure 9 reports the total time to create a hub index for different values of K. We see quadratic growth of both space and time requirements, due to the quadratic growth in the size of a vertex neighborhood. Momentarily we will show that increasing the number of hubs reduces space and time requirements.

Next, we consider how our algorithms scale as the databases grow in size. In Figure 10 we show the total storage required to store the final index when we (again) choose no more than 2.5% of vertices as hubs, for $K = 12$. Note that the storage consumption scales linearly, despite the fact that the large scaled graphs are tightly interconnected. We also observed that the index construction times scaled linearly with data sets, but we do not show the graph here due to lack of space.

In Figure 11, we see that relatively small increases in the number of hubs can dramatically reduce the storage requirements of a hub index. Again, we consider the case where $S = 10$ and $K = 12$. First, notice that if we choose fewer than 0.5% of vertices as hubs, we need significantly more space to store the final index; recall that we degenerate to the self-join algorithm when no hubs are selected. If we can choose up to 5% of vertices as hubs we see that the storage ratio for the final index drops to about 3.93. As we mentioned earlier, the graph shows that our algorithm smoothly transitions into a main-memory shortest-path computation as more memory is made available. Though not displayed here, the index construction time also follows a trend similar to the space requirements.

In general, the index edge-lists are still large enough that any additional compression is useful. By altering our on-disk representation of edge-lists we can gain significant savings: we store a given edge-list as an adjacency list and then use *delta-compression*, a standard technique used in information retrieval systems for compressing sorted data [ZMSD93]. Our experiments showed that when $K = 12$ and at most 2.5% of the vertices are hubs, the final index, including the delta-compressed zapped edge-list, is 2.0 times the size of the initial edge-list; it is 2.5 times the size of the delta-compressed initial edge-list. (As mentioned above, without compression the final index was 6 times larger than the input.) Our index construction algorithms can be easily modified to operate on the delta-compressed edge-lists.

Finally, we give a couple of examples of query execution time. As can be expected, query times vary based on the size of the input sets. Consider yet again the query "*Find* movie *Near* Travolta Cage." In our (unscaled) IMDB dataset, $|Find| \approx 2000$ and $|Near| = 2$. With "movie" objects clustered together and no more than 2.5% of the vertices as hubs, the query takes 1.52 seconds (beyond the *Find/Near* queries executed by Lore). For the query "*Find* movie *Near* location," ($|Find| \approx 2000, |Near| \approx 200$) execution takes 2.78 seconds. Experiments not shown here indicate that choosing more hubs reduces query execution time.

7 Related Work

Most existing approaches for supporting proximity search in databases are restricted to searching only within specific fields known to store unstructured text [Ora97, DM97]. Such approaches do not consider interrelationships between the different fields (unless manually specified through a query). One company, DTL, markets a technology called DataSpot (www.dataspot.com) for plain language search over databases [DGEP98]. DataSpot is also based on a graph model, using heuristics to significantly prune the search space; their specific algorithms have not been made public.

A universal relation [Ull89] is a single relational view of an entire database, which enables users to pose simple queries over relational data. A universal relation brings tuples within close "proximity" together. Still, this approach does not support proximity search in general, and it provides no mechanism for ranking relevant results.

There has been extensive work on the problem of computing the *transitive closure* of a disk-resident directed graph, strictly more general than the problem of computing shortest distances up to some K. Work by Dar and Ramakrishnan [DR94] examines many algorithms for this problem and supplies comparative performance evaluation, as well as discussion of useful measures of performance. In principle, it would be possible to apply these algorithms to our problem, but in practice this cannot be done efficiently. For one, the algorithms are designed to perform transitive closure queries at runtime. An input query is a set of vertices $Q \subseteq V$, and the output is the set of all vertices $R \subseteq V$ reachable from this set. Ullman and Yannakakis [UY91] obtain a bound of $O(N^3/\sqrt{M})$ I/Os for computing the transitive closure of a graph with N nodes and main memory M. The runtime performance hit could be solved by pre-computing the transitive closure and storing it on disk. However, the space required by such a scheme would be huge ($O(V^2)$). Our schemes avoid these pitfalls by not explicitly computing or storing full neighborhoods.

8 Conclusion and Future Work

We have presented a framework for supporting proximity search across an entire database. While traditional IR proximity searches are based on finding keywords in textual documents, we demonstrated a general approach for proximity search over any large set of interconnected data objects. We formalized our notion of proximity and proposed several scoring functions. As an application of our search techniques, we created a system that supports keyword proximity search over databases, yielding interesting and intuitive results. Measuring proximity depends on efficient computation of distances between objects for a disk-based graph. We gave a formal framework and several approaches for solving the problem, focusing on hub indexing. Experiments showed that creating hub indexes is reasonably fast, the indexes are compact, and they can be used to quickly find shortest distances at search time.

For future work, we are considering the following directions.

- We plan to continue to enhance our indexing algorithms. We are investigating improved techniques for selecting hubs, especially when we can determine certain properties of the input graph. In addition, we plan to further investigate techniques for compressing K-neighborhoods on disk. If we could pre-compute all K-neighborhoods (rather than just the "hub-bordered" neighborhoods), we could dramatically improve query time. Without compression, however, the space requirements of such a structure would be enormous. Finally, we are considering the possibility of improving performance by maintaining approximate (rather than exact) distances between objects.

- Users may desire query flexibility beyond basic *Find/Near* queries. For example, someone may want to find the movies that are near Travolta but *not* near Cage. As another example, one might be interested in finding the actors near the movies near Cage: *Find* actor *Near* (*Find* movie *Near* Cage). Implementing such general functionality requires schemes for combining results from multiple proximity searches.

- To enable many applications, we want to integrate proximity search into structured query languages such as SQL. In the relational setting, we anticipate several interesting issues involved in combining traditional relations with ranked tuples that may be returned by the proximity search. Support for ranked tuples was broached by Fagin [Fag96], who suggests using fuzzy sets.

- We are looking at how to incrementally maintain our indexing structures as the underlying database changes.

Acknowledgements

We are grateful to Vineet Gossain and Jason McHugh for their helpful comments.

References

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[AKR93] A. Agrawal, P. Klein, and R. Ravi. Cutting down on fill using nested dissection: provably good elimination orderings. In J. A. George, J. R. Gilbert, and J. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms, IMA Volumes in Mathematics and its applications*, pages 31–55. Springer-Verlag, New York, 1993.

[Bod96] H. Bodlaender. A linear-time algorithm for finding tree-decompositions of small tree-width. *SIAM Journal on Computing*, 25(6):1305–1317, Dec 1996.

[Con97] World Wide Web Consortium. Extensible markup language (XML). http://www.w3.org/ TR/WD-xml-lang-970331.html, December 1997. Proposed recommendation.

[CZ95] S. Chaudhuri and C. Zaroliagis. Shortest paths in digraphs of small treewidth. In Z. Fulop, editor, *Proc. Int. Conference on Automata, Languages and Programming*, pages 244–255, Szeged, Hungary, July 1995.

[DGEP98] S. Dar, S. Geva, G. Entin, and E. Palmon. DTL's DataSpot: Database exploration using plain language. In *Proceedings of the Twenty-Fourth International Conference on Very Large Data Bases*, 1998.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DM97] S. DeBloch and N. Mattos. Integrating SQL databases with content-specific search engines. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, 1997.

[DR94] Shaul Dar and Raghu Ramakrishnan. A performance study of transitive closure algorithms. In *Proceedings of SIGMOD*, pages 454–465, May 1994.

[Fag96] R. Fagin. Combining fuzzy information from multiple systems. In *Proceedings of the Fifteenth Symposium on Principles of Database Systems*, pages 216–226, Montreal, Canada, June 1996.

[Flo62] R. W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

[Goo61] I. J. Good. A causal calculus. *British Journal of the Philosophy of Science*, 11:305–318, 1961.

[GSVGM98] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. (Extended Version). Technical Report, Stanford University. Available at http://www-db.stanford.edu/pub/papers/proximity.ps, 1998.

[HKRS94] M. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. In *26th Annual ACM Symposium on Theory of Computing*, Montreal, Quebec. Canada, May 1994.

[KS] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. *Algorithmica*. Submitted.

[LT80] R. Lipton and R. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, 1980.

[MAG$^+$97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), September 1997.

[Ora97] Oracle Corp. Managing text with Oracle8 ConText cartridge. http://www.oracle.com/st/o8collateral/html/xctx5bwp.html, 1997. White paper.

[Pel97] D. Peleg. Proximity-preserving labelling schemes. Manuscript, 1997.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[Sal89] Gerard Salton. *Automatic Text Processing: The transformation, analysis, and retrieval of information by computer.* Addison-Wesley, 1989.

[Ull89] J. Ullman. *Principles of Database and Knowledge-base systems, Volume II.* Computer Science Press, Rockville, Maryland, 1989.

[UY91] J. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence 3*, pages 331–360, 1991.

[ZMSD93] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. Searching large lexicons for partially specified terms using compressed inverted files. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 290–301, 1993.