Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity

William W. Cohen

AT&T Labs—Research

180 Park Avenue, Florham Park NJ 07932

wcohen@research.att.com

Abstract

Most databases contain "name constants" like course numbers, personal names, and place names that correspond to entities in the real world. Previous work in integration of heterogeneous databases has assumed that local name constants can be mapped into an appropriate global domain by normalization. However, in many cases, this assumption does not hold; determining if two name constants should be considered identical can require detailed knowledge of the world, the purpose of the user's query, or both. In this paper, we reject the assumption that global domains can be easily constructed, and assume instead that the names are given in natural language text. We then propose a logic called WHIRL which reasons explicitly about the *similarity* of local names, as measured using the vector-space model commonly adopted in statistical information retrieval. We describe an efficient implementation of WHIRL and evaluate it experimentally on data extracted from the World Wide Web. We show that WHIRL is much faster than naive inference methods, even for short queries. We also show that inferences made by WHIRL are surprisingly accurate, equaling the accuracy of hand-coded normalization routines on one benchmark problem, and outperforming exact matching with a plausible global domain on a second.

1 Introduction

The integration of distributed, heterogeneous databases, sometimes called data integration, is an active area of research [14; 27; 2; 19; 40; 6]. Largely inspired by the proliferation of database-like sources on the World Wide Web, previous researchers have addressed a diverse set of problems, ranging from access to "semi-structured" information sources [38; 1; 39] to combining databases with differing schemata [26; 13].

In this paper we will consider another aspect of data integration: *the integration of databases that lack common domains.* To illustrate this problem, consider a relation `p` with schema `p(company,industry)` that associates companies with a short description of their industries, and a second relation `q` with schema `q(company,website)` that associates companies with their home pages. If `p` and `q` are taken from different, heterogeneous databases, then the same company might be denoted by different constants x and x' in `p` and `q` respectively, making it impossible to join `p` and `q` in the usual way.

In general, most databases contain many domains in which the individual constants correspond to entities in the real world; examples of such "name domains" include course numbers, personal names, company names, movie names, and place names. Most previous work in data integration either assumes these "name domains" to be global, or else assumes that local "name constants" can be mapped into a global domain by a relatively simple normalization process. However, examination of real-world information sources reveals many cases in which creating a global domain by normalization is difficult. In general, the mapping from "name constants" to real entities can differ in subtle ways from database to database, making it difficult to determine if two constants are co-referent (*i.e.*, refer to the same entity).

For instance, in two Web databases listing educational software companies, we find the name constants "Microsoft" and "Microsoft Kids": do these denote the same company, or not? In another pair of Web sources, the names "Kestrel" and "American Kestrel" appear: do these denote the same type of bird, or not? To take examples from a domain familiar to most readers, under what circumstances should "MIT" and "MIT Media Lab" be considered identical? Finally, which pairs of the following names correspond to the same research institution: "AT&T Bell Labs", "AT&T Labs", "AT&T Labs–Research", "AT&T Research", "Bell Labs", and "Bell Telephone Labs"?

In short, in many real-world data sources—particularly those found on the Web—determining if two name constants are co-referent is far from trivial. Frequently it requires detailed knowledge of the world, the purpose of the user's query, or both. We also note that the problem of common domains is both critical and fundamental: critical, since an inappropriate mapping from local to global domains will lead to erroneous or missing answers to user queries, and fundamental, since all previous techniques for integration of heterogeneous databases require common domains.

In this paper, *we reject the assumption that common domains exist,* or can be easily constructed. Instead, we will assume that the *names assigned to real-world entities are given in natural language text.* Under this assumption, determining if two names are co-referent is a problem of understanding unrestricted natural language, leading immediately to the conclusion that it is impossible determine co-reference reliably. Therefore, determining name co-reference should *not* be handled by automatic means which are hidden to the user.

Instead, we propose a new logic for database integration called WHIRL. WHIRL retains the original local names and reasons explicitly about the *similarity* of pairs of names, using statistical measures of document similarity that have been developed in the information retrieval (IR) community. As in conventional database systems, the answer to a user's query is a set of tuples; however, these tuples are ordered so that the "best" answers are presented to the user first. WHIRL considers tuples to be "better" when the name co-reference conditions required by the user's query are more likely to hold.

WHIRL thus combines some properties of statistical IR systems, and some properties of database systems. Like statistical IR systems, WHIRL reasons about the similarity of documents, and outputs an ordered list of answers. (In statistical IR systems, documents are generally presented in order of estimated relevance to the user's query.) However, like a database system, WHIRL's answers are tuples instead of documents, and WHIRL queries can involve many different relations, instead of a single document collection.

In the remainder of the paper, we will first present the semantics of the WHIRL query language, and then describe an efficient query algorithm for WHIRL. Semantically WHIRL is much like earlier probabilistic or "fuzzy" database logics [18; 4] ; however, certain properties of text make efficient inference a bit trickier. In particular, it is typically the case that many pairs of names will be weakly similar, but few will be strongly similar; this leads to inefficiencies for probabilistic inference algorithms that compute all tuples with non-zero probability. Our query-answering algorithm is novel in that it finds the highest-scoring answer tuples without generating all the low-scoring tuples.

Finally, we evaluate the algorithm experimentally on real-world data extracted from the Web. We show that our algorithm is much faster than naive inference methods, even for short queries. We also show that the inferences of the system are surprisingly accurate, as measured with average precision. In one case WHIRL's performance equals the performance of a hand-constructed, domain-specific normalization routine. In a second case, WHIRL's performance gives better performance than matching on a plausible global domain.

2 Semantics of the WHIRL Query Language

2.1 The vector space representation for documents

As noted above, we will adopt a data model in which real-world entities are named by natural language text. One widely used method for representing text is the *vector space model* [36], which we will now briefly review. We assume a vocabulary T of *terms*, which will be treated as atomic; terms might include words, phrases, or word stems (morphologically derived word prefixes). A fragment of text is represented as *document vector:* a vector of real numbers $\mathbf{v} \in \mathcal{R}^{|T|}$, each component of which corresponds to a term $t \in T$. We will denote the component of $\mathbf{v}$ which corresponds to $t \in T$ by $\mathbf{v}^t$.

A number of schemes have been proposed for assigning weights to terms. We found it convenient to adopt the widely used TF-IDF weighting scheme with unit length normalization. Assuming that the document represented by $\mathbf{v}$ is a member of a document collection C, define $\hat{\mathbf{v}}^t$ to have the value zero if t is not present in the document represented by $\mathbf{v}$, and otherwise the value $\hat{\mathbf{v}}^t = (\log(TF_{\mathbf{v},t})+1)\cdot\log(IDF_t)$, where the "term frequency" $TF_{\mathbf{v},t}$ is the number of times that term t occurs in the document represented by $\mathbf{v}$, and the "inverse document frequency" IDF_t is $\frac{|C|}{|C_t|}$, where C_t is the subset of documents in C that contain the term t. This vector is then normalized to unit length.

The *similarity* of two document vectors $\mathbf{v}$ and $\mathbf{w}$ is given by the formula $sim(\mathbf{v},\mathbf{w}) = \sum_{t\in T} \mathbf{v}^t \cdot \mathbf{w}^t$, which is usually interpreted as the cosine of the angle between $\mathbf{v}$ and $\mathbf{w}$. Since every document vector $\mathbf{v}$ has unit length, $sim(\mathbf{v},\mathbf{w})$ is always between zero and one.

We note in passing that although these vectors are conceptually very long, they are also very sparse: if a document contains only k terms, then all but k components of its vector representation will have zero weight. There are well-known methods for efficiently manipulating these sparse vectors.

The general idea behind this scheme is that the magnitude of the component $\mathbf{v}^t$ is related to the "importance" of the term t in the document represented by $\mathbf{v}$. Two documents are similar when they share many "important" terms. The TF-IDF weighting scheme assigns higher weights to terms that occur *infrequently* in the collection C.[1] In a collection of company names, for instance, common terms like "Inc." and "Ltd." would have low weights; uniquely appearing terms like "Lucent" and "Microsoft" would have high weights; and terms of intermediate frequency like "Acme" and "American" would have intermediate weights.

2.2 Conjunctive queries over relations of documents

We will assume that all data is stored in relations, but that the primitive elements of each relation are document vectors, rather than atoms. We call this data model STIR, for Simple Texts In Relations—"simple" emphasizing the fact that the texts are assumed to have no additional structure.

More precisely, an *extensional database (EDB)* consists of a term vocabulary T and set of relations $\{p_1,\ldots,p_n\}$. Associated with each relation p is a set of tuples $tuples(p)$. Every tuple $\langle \mathbf{v}1,\ldots,\mathbf{v}\mathrm{k}\rangle \in tuples(p)$ has exactly k components, and each of these components $\mathbf{v}\mathrm{i}$ is a document vector over T. We will also assume that a *score* is associated with every tuple in p. This score will always be between zero and one, and will be denoted $score(\langle \mathbf{v}1,\ldots,\mathbf{v}\mathrm{k}\rangle \in tuples(p))$. In most applications, the score of every tuple in a base relation will be one; however, it will be convenient to allow non-unit scores, so that materialized views can be stored.

WHIRL (for Word-based Heterogeneous Information Retrieval Logic) is a query language for accessing these relations. A *conjunctive WHIRL query* is written $B_1 \wedge \ldots \wedge B_k$ where each B_i is a *literal*. There are two types of literals. An *EDB literal* is written $p(X_1,\ldots,X_k)$ where p is the name of an EDB relation, and the X_i's are *variable symbols* (or simply *variables*). A *similarity literal* is written $X \sim Y$, where X and Y are variables; intuitively, this will be interpreted as a requirement that documents X and Y be similar. We will henceforth assume that if X appears in a similarity literal in a query Q, then X also appears in some EDB literal in Q.

Example 1 To return to the example of the introduction, the join of the relations p and q might be approximated by the query Q_1:

```
p(Company1,Industry) ∧ q(Company2,WebSite)
∧ Company1~Company2
```

(Note that this is different from an equijoin of `p` and `q`, which could be written `p(Company,Industry) ∧ q(Company,WebSite)`.) To find Web sites for companies in

[1]The weighting scheme also gives higher weights to terms that occur *frequently* in a document. However, in this context, this heuristic is probably not that important, since names are usually short enough so that each term occurs only once.

the telecommunications industry one might use the query Q_2:

```
p(Company1,Industry) ∧ q(Company2,WebSite)
∧ Company1~Company2 ∧ const1(IO)
∧ Industry~IO
```

where the relation `const1` contains a single document describing the industry of interest, such as "telecommunications equipment and/or services".

To define the semantics of WHIRL, we will extend the notion of score to single literals, and then to conjunctions. Let B be a literal, and θ a substitution such that $B\theta$ is ground. If B is an EDB literal $p(X_1, \ldots, X_k)$, we define $score(B\theta) = score(\langle X_1\theta, \ldots, X_k\theta\rangle \in p)$ if $\langle X_1\theta, \ldots, X_k\theta\rangle \in tuples(p)$, and $score(B\theta) = 0$ otherwise. If B is a similarity literal $X \sim Y$, we define $score(B\theta) = sim(X\theta, Y\theta)$.

Now, if $Q = B_1 \wedge \ldots \wedge B_k$ is a query and $Q\theta$ is ground, we define $score(Q\theta) = \prod_{i=1}^{n} score(B_i\theta)$. In other words, we score conjunctive queries by combining the scores of literals as if they were independent probabilities.[2]

Recall that the answer to a conventional conjunctive query is the set of ground substitutions that make the query "true" (*i.e.*, provable against the EDB). In WHIRL, the notion of provability has been replaced with the "soft" notion of score: substitutions with a high score are intended to be better answers than those with a low score. It seems reasonable to assume that users will be most interested in seeing the high-scoring substitutions, and will be less interested in the low-scoring substitutions. We formalize this as follows. Given an EDB, we define the *full answer set* S_Q for a conjunctive query Q to be the set of all θ_i such that $Q\theta_i$ is ground and has a non-zero score. We define an *r-answer* R_Q for a conjunctive query Q to be an ordered list of substitutions $\theta_1, \ldots, \theta_r$ from the full answer set S_Q such that:

- for all $\theta_i \in R_Q$ and $\sigma \in S_Q - R_Q$, $score(Q\theta_i) \geq score(Q\sigma)$; and
- for all $\theta_i, \theta_j \in R_Q$ where $i < j$, $score(Q\theta_i) \geq score(Q\theta_j)$.

In other words, R_Q contains r highest-scoring substitutions, ordered by non-increasing score.

We will assume the output of a query-answering algorithm given the query Q will *not* be a full answer set, but rather an r-answer for Q, where r is a parameter fixed by the user. To motivate the notion of an r-answer, observe that in typical situations the full answer set for WHIRL queries will be very large. For example, the full answer set for the query Q_1 given as an example above would include all pairs of company names `Company1`, `Company2` that both contain the term "Inc". This set might be very large. Indeed, if we assume that a fixed fraction $\frac{1}{k}$ of company names contain the term "Inc", and that p and q each contain a random selection of n company names, then one would expect the size of the full answer set to contain $(\frac{n}{k})^2$ substitutions simply due to the matches on the term "Inc"; further the full answer set for the join of m relations of this sort would be of size at least $(\frac{n}{k})^m$.

To further illustrate this point, we computed the pairwise similarities of two lists p and q of company names,[3] with p containing 1163 names, q containing 976 names. Although the intersection of p and q appears to contain only about 112 companies, over 314,000 name pairs had non-zero similarity. In this case, the number of non-zero similarities can be greatly reduced by discarding a few very frequent terms like "Inc".[4] However, even *after* this preprocessing, there are more than 19,000 non-zero pairwise similarities—more than 170 times the number of correct pairings. This is due to a large number of moderately frequently terms (like "American" and "Airlines") that cannot be safely discarded.

In conclusion, it is in general impractical to compute full answer sets for complex queries, and antisocial to present them to a user. This leads to the assumption of an r-answer.

2.3 Unions of conjunctive queries

The scoring scheme given above for conjunctive queries can be fairly easily extended to certain more expressive languages. Below we consider one such extension, which corresponds to projections of unions of conjunctive queries.

A *basic WHIRL clause* is written $p(X_1, \ldots, X_k) \leftarrow Q$, where Q is a conjunctive WHIRL query that contains all of the X_i's. A *basic WHIRL view* $\mathcal{V}$ is a set of basic WHIRL clauses with heads that have the same predicate symbol p and arity k. Notice that by this definition, all the literals in a clause body are either EDB literals or similarity literals—in other words, the view is "flat", involving only extensionally defined predicates.

Now, consider a ground instance $a = p(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ of the head of some view clause. We define the *support of a (relative to the view $\mathcal{V}$ and a given EDB)* to be the set of triples $\langle A \leftarrow Q, \theta, s\rangle$ satisfying these conditions:

1. $(A \leftarrow Q) \in \mathcal{V}$; and
2. $A\theta = a$, and $Q\theta$ is ground; and
3. $score(Q\theta) = s$, and $s > 0$.

The support of a will be written $support(a)$. We then define the score of $\langle \mathbf{x}_1, \ldots, \mathbf{x}_k\rangle$ in p as follows:[5]

$$score(\langle \mathbf{x}_1, \ldots, \mathbf{x}_k\rangle \in p) = 1 - \prod_{\langle C,\theta,s\rangle \in support(p(\mathbf{X}_1,\ldots,\mathbf{X}_k))} (1-s) \tag{1}$$

We can now define the *materialization of the view $\mathcal{V}$* to be a relation with name p which contains all tuples $\langle \mathbf{x}_1, \ldots, \mathbf{x}_k\rangle$ such that $score(\langle \mathbf{x}_1, \ldots, \mathbf{x}_k\rangle \in p) > 0$.

Unfortunately, while this definition is natural, there is a difficulty with using it in practice. In a conventional setting, it is easy to materialize a view of this sort, given a mechanism for solving a conjunctive query. In WHIRL, we would prefer to assume only a mechanism for computing r-answers to conjunctive queries. However, since Equation 1 involves a support set of unbounded size, it appears that a r-answers are not enough to even score a single ground instance a.

Fortunately, however, low-scoring substitutions have only a minimal impact on the score of a. Specifically, if $\langle C, \theta, s\rangle$ is such that s is close to zero, then the corresponding factor of $(1 - s)$ in the score for a is close to one. One can thus approximate the score of Equation 1 using a smaller

[2] Of course, similarity scores are not independent probabilities, so there is no reason to expect this combination method to be optimal. However, this combination method is simple and relatively well-understood, and is in our view a reasonable starting point for research on this sort of data integration system.

[3] These lists are the relations `HooverWeb` and `Iontech` from Table 1 below.

[4] In fact, certain common "stop words" are generally discarded in statistical IR systems.

[5] As a brief motivation for this formula, note that it is some sense a dual of multiplication: if e_1 and e_2 are independent probabilistic events with probability p_1 and p_2 respectively, then the probability of $(e_1 \wedge e_2)$ is $p_1 \cdot p_2$, and the probability of $(e_1 \vee e_2)$ is $1-(1-p_1)(1-p_2)$.

set of high-scoring substitutions, such as those found in an r-answer for moderately large r.

In particular, let $\mathcal{V}$ contain the clauses $A_1 \leftarrow Q_1, \ldots, A_n \leftarrow Q_n$, let $R_{Q_1}, \ldots, R_{Q_n}$ be r-answers for the Q_i's, and let $R = \cup_i R_{Q_i}$. Now define the *r-support for a from R* to be the set

$$\{\langle A \leftarrow Q, \theta, s\rangle : \langle A \leftarrow Q, \theta, s\rangle \in \mathit{support}(a) \text{ and } \theta \in R\}$$

Also define the *r-score for a from R* by replacing $\mathit{support}(a)$ in Equation 1 with the r-support set for a. Finally, define the *r-materialization of $\mathcal{V}$ from R* to contain all tuples $\mathbf{x}_1, \ldots, \mathbf{x}_k$ with non-zero r-score, with the score of $\mathbf{x}_1, \ldots, \mathbf{x}_k$ in p being its r-score from R.

Clearly, the r-materialization of a view can be constructed using only an r-answer for each clause body involved in the view. As r is increased, the r-answers will include more and more high-scoring substitutions, and the r-materialization will become a better and better approximation to the full materialized view. Thus given an efficient mechanism for computing r-answers for conjunctive views, one can efficiently approximate the answers to more complex queries.

2.4 Relation to other logics

At the level described so far, WHIRL is closely related to earlier formalisms for probabilistic databases. In particular, if similarities were stored in a relation `sim(X,Y)` instead of being computed "on the fly", and certain irredundancy assumptions are made, then WHIRL is a strict subset of Fuhr's probabilistic Datalog [18].[6] There are also close connections to existing formalisms for probabilistic relational databases [4] .

Given this, it might well be asked why it is necessary to introduce a new and more restricted probabilistic logic. Our response is that the assumptions made in WHIRL enable relatively efficient inference, without making the logic too restricted to handle its intended task: integration of heterogeneous, autonomous databases by reasoning about the similarity of names. In particular, these restrictions make it possible to generate an r-answer for conjunctive queries efficiently, even if the full answer set is large, and even if the document vectors used to represent local entity names are quite diverse. These claims will be substantiated more fully in Section 4 below.

3 The Query Processing Algorithm

3.1 Overview of the algorithm

The current implementation of WHIRL implements the operations of finding the r-answer to a query and the r-materialization of a view.[7] In this section we will describe an efficient strategy for constructing an r-answer to a query, and then present some detailed examples of the algorithm. First, however, we will give a short overview of the main ideas used in the algorithm.

[6] Specifically, if one assumes that queries $B_1 \wedge \ldots B_k$ are "irredundant" in the sense that there is no ground substitution θ with non-zero score such that $B_i\theta = B_j\theta$ for $i \neq j$, and also make the same independence assumptions made in Fuhr's Datalog_{PID}, then the score for a WHIRL predicate is exactly the probability of the corresponding compound event, which is the same as the probability computed by Datalog_{PID}.

[7] However, note that not every relational algebra query can be expressed in WHIRL, since there is no equivalent of negation or set difference. Nor are SQL operations like grouping and sorting supported.

In WHIRL, finding an r-answer is viewed as an optimization problem; in particular, the query processing algorithm uses a general method called A$*$ search [33; 25] to find the highest-scoring r substitutions for a query. Viewing query processing as search is natural, given that the goal is to find a *small* number of *good* substitutions, rather than all satisfying substitutions; the search method we use also generalizes certain techniques used in IR ranked retrieval [41]. However, using search in query processing is unusual for database systems, which more typically use search only in optimizing a query.

To motivate our use of search, consider finding an r-answer to the WHIRL query

```
insiderTip(X) ∧ publiclyTraded(Y) ∧ X~Y
```

where the relation `publiclyTraded` is very large, but the relation `insiderTip` is very small. In processing the corresponding equijoin `insiderTip(X) ∧ publiclyTraded(Y) ∧ X=Y` with a conventional database system, one would first construct a query plan: for example, one might first find all bindings for X, and then use an index to find all values Y in the first column of `publiclyTraded` that are equivalent to some X. It is tempting to extend such a query plan to WHIRL, by simply changing the second step to find all values Y that are similar to some X.

However, this natural extension can be quite inefficient. Imagine that `insiderTip` contains the vector $\mathbf{x}1$, corresponding to the document "Armadillos, Inc". Due to the frequent term "Inc", there will be many documents Y that have non-zero similarity to $\mathbf{x}1$, and it will be expensive to retrieve all of these documents Y and compute their similarity to $\mathbf{x}1$.

One way of avoiding this expense is to start by retrieving a *small* number of documents Y that are likely to be highly similar to $\mathbf{x}1$. In this case, one might use an index to find all Y's that contain the rare term "Armadillos". Since "Armadillos" is rare, this step will be inexpensive, and the Y's retrieved in this step must be somewhat similar to $\mathbf{x}1$. (Recall that the weight of a term depends inversely on its frequency, so rare terms have high weight, and hence these Y's will share at least one high-weight term with X.) Conversely, any Y' *not* retrieved in this step must be somewhat *dissimilar* to $\mathbf{x}1$, since such a Y' *cannot* share with $\mathbf{x}1$ the high-weight term "Armadillos". This suggests that if r is small, and an appropriate pruning method is used, a subtask like "find the r documents Y that are most similar to $\mathbf{x}1$" might be accomplished efficiently by the subplan of "find all Y's containing the term 'Armadillos' ".

Of course, this subplan depends on the vector $\mathbf{x}1$. To find the Y's most similar to the document "The American Software Company" (in which every term is somewhat frequent) a very different type of subplan might be required. The observations suggest that query processing should proceed in small steps, and that these steps should be scheduled dynamically, in a manner that depends on the specific document vectors being processed.

In the query processing algorithm described below, we will search through a space of partial substitutions: for example, one state in the search space for the query given above would correspond to the substitution that maps X to $\mathbf{x}1$ and leaves Y unbound. The steps we take through this search space are small ones, as suggested by the discussion above; for instance, one operation is to select a single term t and use an inverted index to find plausible bindings for a single unbound variable. Finally, we allow the search algorithm to order these operations dynamically, focusing on those partial substitutions that seem to be most promising, and effectively pruning partial substitutions that cannot lead to a high scoring ground substitution.

3.2 A$*$ search

A$*$ search (summarized in Figure 1) is a graph search method which attempts to find the highest scoring path between a given *start state* s_0 and a *goal state* [33; 25]. Goal states are defined by a `goalState` predicate. The graph being searched is defined by a function `children`(s), which returns the set of states directly reachable from state s. To conduct the search the $A*$ algorithm maintains a set `OPEN` of states that might lie on a path to some goal state. Initially `OPEN` contains only the start state s_0. At each subsequent step of the algorithm, a single state is removed from the `OPEN` set; in particular, the state s that is "best" according to a *heuristic function*, $h(s)$, is removed from `OPEN`. If s is a goal state, then this state is output; otherwise, all children of s are added to the `OPEN` set. The search continues until r goal states have been output, or the search space is exhausted.

The procedure described above is a variant of the A$*$ procedure normally studied, but it has similar desirable properties. In particular, define a heuristic function $h(\cdot)$ to be *admissible* iff for all states s and all states s' reachable from s, $h(s) \geq h(s')$. Define a graph G to be a *bounded tree* if it is a tree of finite depth in which the goal states are all leaves. It is straightforward to show that if $h(\cdot)$ is admissible and the graph G defined by the `children` function is a bounded tree containing at least r goal states, then this A$*$ variant will output in non-increasing order the r goal states with the largest heuristic values. Thus the A$*$ search will compute an r-answer for Q, whenever the conditions above are met, each state s encodes a substitution θ_s, and $h(s) = score(Q\theta_s)$ for ground θ_s.

3.3 The operators and heuristic function

We will now explain how this general search method has been instantiated in WHIRL. We will assume that in the query Q, each variable in Q appears exactly once in an EDB literal.[8] In processing queries, the following data structures will be used. An *inverted index* will map terms $t \in T$ to the tuples that contain them: specifically, we will assume a function `index`(t,p,i) which returns the set of tuples $\langle \mathbf{v}_1, \ldots, \mathbf{v}_i, \ldots, \mathbf{v}_k \rangle$ in *tuples*(p) such that $\mathbf{v_i}^t > 0$. We will also precompute the function `maxweight`(t,p,i), which returns the maximum value of $\mathbf{v_i}^t$ over all documents $\mathbf{v_i}$ in the i-th column of p.

The states of the graph searched will be pairs $\langle \theta, E \rangle$, where θ is a substitution, and E is a set of *exclusions*. Goal states will be those for which θ is ground for Q, and the initial state s_0 is $\langle \emptyset, \emptyset \rangle$. An *exclusion* is a pair $\langle t, Y \rangle$ where t is a term and Y is a variable. Intuitively, it means that the variable Y must not be bound to a document containing the term t. Formally, we will say that a substitution θ is *E-valid* if $\forall \langle t, Y \rangle \in E$, $(Y\theta)^t = 0$. Below we will define the `children` function so that all descendents of a node $\langle s, E \rangle$ must be E-valid; by making appropriate use of these exclusions we will force the graph defined by the `children` function to be a tree.

We will adopt the following terminology. Given a substitution θ and query Q, a similarity literal $X \sim Y$ is *constraining* iff exactly one of $X\theta$ and $Y\theta$ are ground. Without loss of generality, we assume that $X\theta$ is ground and $Y\theta$ is not. For any variable Y, the EDB literal of Q that contains Y is the *generator*[9] for Y, the position ℓ of Y within this literal is Y's *generation index*.

Children are generated in two ways: by *exploding* a state, or by *constraining* a state. *Exploding* a state corresponds to picking all possible bindings of some unbound EDB literal. To explode a state $s = \langle \theta, E \rangle$, pick some EDB literal $p(Y_1, \ldots, Y_k)$ such that all the Y_i's are unbound by θ, and then construct all states of the form $\langle \theta \cup \{Y_1 = \mathbf{v}1, \ldots, Y_k = \mathbf{v}\mathrm{k}\}, E \rangle$ such that $\langle \mathbf{v}_1, \ldots, \mathbf{v}_k \rangle \in tuples(p)$ and $\theta \cup \{Y_1 = \mathbf{v}1, \ldots, Y_k = \mathbf{v}\mathrm{k}\}$ is E-valid. These are the children of s.

The second operation of *constraining* a state implements a sort of sideways information passing. To *constrain* a state $s = \langle \theta, E \rangle$, pick some constraining literal $X \sim Y$ and some term t with non-zero weight in the document $X\theta$ such that $\langle t, Y \rangle \notin E$. Let $p(Y_1, \ldots, Y_k)$ be the generator for the (unbound) variable Y, and let ℓ be Y's generation index. Two sets of child states will now be constructed. The first is a singleton set containing the state $s' = \langle \theta, E' \rangle$, where $E' = E \cup \{\langle t, Y \rangle\}$. Notice that by further constraining s', other constraining literals and other terms t in $X\theta$ can be used to generate plausible variable bindings. The second set S_t contains all states $\langle \theta_i, E \rangle$ such that $\theta_i = \theta \cup \{Y_1 = \mathbf{v}1, \ldots, Y_k = \mathbf{v}\mathrm{k}\}$ for some $\langle \mathbf{v}_1, \ldots, \mathbf{v}_k \rangle \in$ `index`(t, p, ℓ) and θ_i is E-valid. The states in S_t thus correspond to binding Y to some vector containing the term t. The set `children`(s) is $S_t \cup \{s'\}$.

It is easy to see that if s_i and s_j are two different states in S_t, then their descendents must be disjoint. Furthermore, the descendents of s' must be disjoint from the descendents of any $s_i \in S_t$, since all descendents of s' are valid for E', and none of the descendents of s_i can be valid for E'. Thus the graph generated by this `children` function is a tree.

Given the operations above, there will typically be many ways to "constrain" or "explode" a state. In the current implementation of WHIRL, a state is always constrained using the pair $\langle t, Y \rangle$ such that $\mathbf{x}^t \cdot$ `maxweight`(t, p, ℓ) is maximal (where p and ℓ are the generator and generation index for Y.) States are exploded only if there are no constraining literals, and then always exploded using the EDB relation containing the fewest tuples.

It remains to define the heuristic function. (For convenience, we will use $h(\theta, E)$ for $h(\langle \theta, E \rangle)$ below.) Recall that the heuristic function $h(\theta, E)$ must be admissible, and must coincide with the scoring function $score(Q\theta)$ on ground substitutions. This implies that $h(\theta, E)$ must be an upper bound on $score(q)$ for any ground instance q of $Q\theta$. We thus define $h(\theta, E)$ to be $\prod_{i=1}^{k} h'(B_i, \theta, E)$, where h' will be an appropriate upper bound on $score(B_i\theta)$. We will let this bound equal $score(B_i\theta)$ for ground $B_i\theta$, and let it equal 1 for non-ground B_i, with the exception of *constraining literals*. For constraining literals, $h'(\cdot)$ is defined as follows:

$$h'(B_i, \theta, E) \equiv \sum_{t \in T : \langle t, Y \rangle \notin E} \mathbf{x}^t \cdot \mathtt{maxweight}(t, p, \ell) \qquad (2)$$

where p and ℓ are the generator and generation index for Y. Note that this is an upper bound on the score of $B_i\sigma$ relative to any ground superset σ of θ that is E-valid.

[8] This restriction is made innocuous by an additional predicate `eq`(X, Y) which is true when X and Y are bound to the same document vector. The implementation of the `eq` predicate is relatively straightforward, and will be ignored in the discussion below.

[9] Notice that for well-formed queries, there will be only one generator for a variable Y.

Generic A∗ search

```
procedure A*(r,s_0,goalState(·),children(·))
begin
    OPEN := {s_0}
    while (OPEN≠∅) do
        s := argmax_{s'∈OPEN} h(s')
        OPEN := OPEN - {s}
        if goalState(s) then
            output ⟨s,h(s)⟩
            exit if r answers printed
        else
            OPEN := OPEN ∪ children(s)
        endif
    endwhile
end
```

As used in WHIRL

Initial state s_0: $\langle\emptyset,\emptyset\rangle$

goalState($\langle\theta,E\rangle$): true iff $Q\theta$ is ground

children($\langle\theta,E\rangle$): see text

$h(\langle\theta,E\rangle)$: $\prod_{i=1} h'(B_i\theta)$ where
$h'(B_i\theta) = score(B_i\theta)$ for ground $B_i\theta$
$h'((X \sim Y)\theta) =$
$\sum_{t\in T:\langle t,Y\rangle\notin E} \mathbf{x}^t \cdot \mathtt{maxweight}(t,p,\ell)$
where $X\theta = \mathbf{x}$, Y is unbound in θ with generator p and generation index ℓ (see text)

Figure 1: Implementation of WHIRL

3.4 Additional details

In the current implementation of WHIRL, the terms of a document are stems produced by the Porter stemming algorithm [34]. In general, the term weights for a document $\mathbf{v_i}$ are computed relative to the collection C of all documents appearing in the i-th column of p. However, the TF-IDF weighting scheme does not provide sensible weights for relations that contain only a single tuple. (These relations are used as a means of introducing "constant" documents into a query.) Therefore weights for these relations must be calculated as if they belonged to some other collection C'.

To set these weights, every query is checked before invoking the query algorithm to see if it contains any EDB literals $p(X_1,\ldots,X_k)$ for a singleton relation p. If one is found, the weights for the document $\mathbf{x_i}$ to which a variables X_i will be bound are computed using the collection of documents found in the column corresponding to Y_i, where Y_i is some variable that appears in a similarity literal with X_i. If several such Y_i's are found, one is chosen arbitrarily. If X_i does not appear in any similarity literals, then its weights are irrelevant to the computation.

The current implementation of WHIRL keeps all indices and document vectors in main memory, and consists of about 5500 lines of C and C++.[10]

3.5 Examples of WHIRL

We will now walk through some examples of this procedure. For clarity, we will assume that terms are words.

Example 2 Consider the query

`const1(IO) ∧ p(Company,Industry) ∧ Industry∼IO`

where `const1` contains the single document "telecommunications services and/or equipment". With $\theta=\emptyset$, there are no constraining literals, so the first step in answering this query will be to explode the smallest relation, in this case `const1`. This will produce one child, s_1, containing the appropriate binding for `IO`, which will be placed on the `OPEN` list.

Next s_1 will be removed from the `OPEN` list. Since `Industry∼IO` is now a constraining literal, a term from the bound variable `IO` will be picked, probably the relatively rare stem "`telecommunications`". The inverted index will be used to find all tuples $\langle \mathbf{co}_1, \mathbf{ind}_1\rangle,\ldots,\langle \mathbf{co}_n, \mathbf{ind}_n\rangle$ such that $\mathbf{ind}_i$ contains the term "`telecommunications`", and n child substitutions that map `Company=co`$_i$ and `Industry=ind`$_i$ will be constructed. Since these substitutions are ground, they will be given $h(\cdot)$ values equal to their actual scores when placed on the `OPEN` list. A new state s_1' containing the exclusion ⟨`telecommunications`, `Industry`⟩ will also be placed on the `OPEN` list. Note that $h(s_1') < h(s_1)$, since the best possible score for the constraining literal `Industry∼IO` can match at most only four terms: "`services`" "`and`", "`or`", "`equipment`", all of which are relatively frequent, and hence have low weight.

Next, a state will again be removed from the `OPEN` list. It may be that $h(s_1')$ is less than the $h(\cdot)$ value of the best goal state; in this case, a ground substitution will be removed from `OPEN`, and an answer will be output. Or it may be that $h(s_1')$ is higher than the best goal state, in which case it will be removed and a new term, perhaps "`equipment`", will be used to generate some additional ground substitutions. These will be added to the `OPEN` list, along with a state s_1'' which has large exclusion set and thus a lower $h(\cdot)$ value.

This process will continue until r documents are generated. Note that it is quite likely that low weight terms such as "`or`" will not be used at all.

In a survey article, Turtle and Flood [41] review a number of query optimization methods for ranked retrieval IR systems. The most effective of these was one they call the *maxscore* optimization. It can be shown that the behavior of WHIRL on queries of the sort shown above is identical to the behavior of an IR system using the *maxscore* optimization.

Example 3 Consider the query

```
p(Company1,Industry) ∧ q(Company2,WebSite)
∧ Company1∼Company2
```

In solving this query, the first step will be to explode the smaller of these relations. Assume that this is `p`, and that `p` contains 1000 tuples. This will add 1000 states $s_1,\ldots,s_{1000}$ to the `OPEN` list. In each of these states, `Company1` and `Industry` are bound, and `Company1∼Company2` is a constraining literal. Thus each of these 1000 states is analogous to the state s_1 in the preceding example.

However, the $h(\cdot)$ values for the states $s_1,\ldots,s_{1000}$ will not be equal. The value of the state s_i associated with the substitution θ_i will depend on the maximum possible score for the literal `Company1∼Company2`, and this will be large only if the high-weight terms in the document `Company1`θ_i

[10] Although it would have been preferable to implement both STIR and WHIRL using MIX [23].

appear in the company field of q. As an example, a one-word document like "3Com" will have a high $h(\cdot)$ value if that term appears (infrequently) in the company field of q, and a zero $h(\cdot)$ value if it does not appear; similarly, a document like "Agents, Inc" will have a low $h(\cdot)$ value if the term "agents" does not appear in the first column of q.

The result is that the next step of the algorithm will be to choose a *promising* state s_i from the OPEN list—a state that could result in an good final score. A term from the Company1 document in s_i—say "3Com"—will then be picked and used to generate bindings for Company2 and WebSite. If any of these bindings results in perfect match, then an answer can be generated on the next iteration of the algorithm.

In short, the operation of WHIRL is somewhat similar to time-sharing 1000 simpler queries on a machine for which the basic unit of computation is to access a single inverted index. However, WHIRL's use of the $h(\cdot)$ function will schedule the computation of these queries in an intelligent way: queries unlikely to produce good answers can be discarded, and low-weight terms are unlikely to be used.

Example 4 Consider the query

```
p(Company1,Industry) ∧ q(Company2,WebSite)
∧ Company1~Company2 ∧ const1(IO) ∧ Industry~IO
```

where the relation const1 contains the single document, "telecommunications and/or equipment". In solving this query, WHIRL will first explode const1 and generate a binding for IO. The literal Industry~IO then becomes constraining, so it will be used to pick bindings for Company1 and Industry using some high-weight term, perhaps "telecommunications".

At this point there will be two types of states on the OPEN list. There will be one state s' in which only IO is bound, and ⟨telecommunications, Industry⟩ is excluded. There will also be several states $s_1, \ldots, s_n$ in which IO, Company1 and Industry are bound; in these states, the literal Company1~Company2 is constraining. If s' has a higher score than any of the s_i's, then s' will be removed from the OPEN list, and another term from the literal Industry~IO will be used to generate additional variable bindings.

However, if some s_i literal has a high $h(\cdot)$ value then it will be taken ahead of s'. Note that this possible when the bindings in s_i lead to a good actual similarity score for Industry~IO as well as a good potential similarity score for Company1~Company2 (as measured by the $h'(\cdot)$ function). If an s_i is picked, then bindings for Company2 and WebSite will be produced, resulting a ground state. This ground state will be removed from the OPEN list on the next iteration only if its $h(\cdot)$ value is higher that of s' and all of the remaining s_i's.

This example illustrates how bindings can be propagated through similarity literals. The binding for IO is first used to generate bindings for Company1 and Industry, and then the binding for Company1 is used to bind Company2 and Website. Note that bindings are generated using high-weight, low-frequency terms first, and low-weight, high-frequency terms only when necessary.

Example 5 We observe that if all scores of all tuples in the EDB are equal to one, and if all documents contain a single term, then the score of all substitutions will be either zero or one. In this case, the full answer set for a WHIRL query Q corresponds exactly to the substitutions that satisfy the query obtained replacing every similarity literal $X \sim Y$ with an equality literal equal(X,Y) (or equivalently, resolving against the unit clause $X \sim X \leftarrow$). Let us consider computing the full answer set (or equivalently, an r-answer for some very large r) for a query of the form

$$p_1(X_1) \ \wedge \ \ldots \ \wedge \ p_k(X_k) \ \wedge \ X_1 \sim X_2$$
$$\wedge \ X_2 \sim X_3 \ \wedge \ \ldots \ \wedge \ X_{k-1} \sim X_k$$

over a such a "conventional" EDB. We will also assume that there is no duplication of documents within a single p_i; in other words, the query corresponds to a k-way join using unique keys.

Although the query algorithm is designed for quite different problems, it is instructive to examine its behavior on this sort of "conventional" k-way join. We claim that in this case, the running time for the WHIRL query algorithm is $O(k \cdot n_i \log n_i)$, where n_i is the size of the smallest relation p_i. To see this, consider the operation of the algorithm on such a query.

The first step for the algorithm will be to "explode" the smallest relation p_i. This will place n_i states $s_1, \ldots, s_{n_i}$ on the OPEN list, where each s_j contains a substitution θ_j binding X_i to a different document xj.

Now consider some state s_j. State s_j contains at most[11] two constraining literals, $B_1 = X_i \sim X_{i+1}$ and $B_2 = X_{i-1} \sim X_i$, and $h(s_j) = h'(B_1, \theta_j, \emptyset) \cdot h'(B_2, \theta_j, \emptyset)$. For this sort of "conventional" data, $h'(B_1, \theta_j, \emptyset)$ is easy to interpret: Equation 2 will evaluate to 1 for B_1 (respectively B_2) only when the single term t contained in xj is contained in some document in p_{i+1} (respectively p_{i-1}) and zero otherwise. So WHIRL will pick some s_j from the OPEN list that binds xj to a value appearing in both p_{i-1} and p_{i+1}, and constrain this state by picking a value for either X_{i-1} or X_{i+1}. In either case, the unique possible binding for the newly constrained variable will be computing in constant time using the index function, s_j will be removed from the OPEN list, and a single new state s'_j will be placed on the OPEN list in its stead. This process can now be repeated for state s'_j, leading eventually to either a goal state descendent of s_j, or to a "dead end" state with an $h(\cdot)$ value of zero.

Following this argument, it is easy to see that the OPEN list never contains more than n_i elements, and that the depth of every state (in the graph defined by the children function) is at most k. This means that at most $n_i \cdot k$ iterations of the while loop of Figure 1 are possible, and leads to a running time of $O(k \cdot n_i \log n_i)$. The final factor of $\log n_i$ bounds the time required to remove a state from an OPEN list of size n_i—although it is unnecessary for this sort of "conventional" data, in general the OPEN list is implemented as a heap.

4 Experimental Results

To evaluate WHIRL, we used the relations described in Table 1. Most of these relations are from Web sites that are plausible subjects for data integration.

We evaluated our implementation of WHIRL along two dimensions. First, we wished to measure the time needed to evaluate queries, and compare this time cost with other strategies. Second, we wished to measure the accuracy of the answers produced by WHIRL. In this evaluation we used the measures of precision and recall traditionally used in the statistical IR community.

All experiments were performed using a prototype implementation of WHIRL, which keeps all indices and document vectors in main memory.

[11]If the smallest relation is p_1 or p_k, then there will be only one constraining literal, but a similar argument applies.

#Tuples	Schema	Source
31,281	`IMDB(movieName,year)`	http://us.imdb.com
37,572	`VideoFlicks(movieName,year,genre)`	http://www.videoflicks.com
232	`Review(movieName,newspaper,review)`	http://www.cinema.pgh.pa.us/movie/reviews
78	`MovieLink(movieName,cinemaName,address,phone,zipcode)`	http://www.movielink.com
2,474	`Hoovers(companyName,industry)`	http://www.hoovers.com
1,163	`HooversWeb(companyName,industry,website)`	http://www.hoovers.com
976	`Iontech(companyName,website,tickertape,industry)`	http://www.iontech.com
13,625	`ReutersTrain(story, keywords)`	http://www.research.att.com/~lewis/reuters21578.html
6,188	`ReutersTest(story, keywords)`	http://www.research.att.com/~lewis/reuters21578.html
990	`Animal1(commonName,scientificName)`	http://endeavor.des.ucdavis.edu/nps
4,719	`Animal2(commonName,scientificName,Range)`	http://www.nceet.snre.umich.edu/EndSpp/ES.lists.html

Table 1: Relations used in the experiments

4.1 Timing results

We evaluated run-time performance with CPU time measurements on a specific class of queries, which we will henceforth call *similarity joins*. A *similarity join* is a query of the form

$$\mathtt{p}(X_1,\ldots,X_i,\ldots,X_k)$$
$$\wedge\ \mathtt{q}(Y_1,\ldots,Y_j,\ldots,Y_b)\ \wedge\ X_i \sim Y_j$$

An answer to this query will consist of the r tuples from p and q such that X_i and Y_j are most similar.

This type of query has several advantages for benchmarking purposes. It is highly relevant to our research goals, since it is directly related to the sort of data integration problem which led us to develop WHIRL. This class of queries is sufficiently constrained in form so that it can be handled using simple algorithms built on top of well-known, previously existing IR search methods. This makes it possible to compare the query optimizations used in WHIRL with previous query optimizations. In particular, we will compare WHIRL with the following algorithms.

- The *naive method for similarity joins* takes each document in the i-th column of relation p in turn, and submits it as a IR ranked retrieval query to a corpus corresponding to the j-column of relation q. The top r results from each of these IR queries are then merged to find the best r pairs overall. This might be more appropriately be called a "semi-naive" method; on each IR query, we use inverted indices, but we employ no special query optimizations.
- As noted above, WHIRL is closely related to *maxscore* optimization [41]. We thus compared WHIRL to a *maxscore method for similarity joins*; this method is analogous to the naive method described above, except that the *maxscore* optimization is used in finding the best r results from each "primitive" query.

To see how these algorithms behave, we used them to compute the top 10 answers[12] for the similarity join of subsets of the `IMDB` and `VideoFlicks` relations. In particular, we joined size n subsets of both relations, for various values of n between 2000 and 30,000. The results are shown in Figure 2. For this data, WHIRL speeds up the *maxscore* method by a factor of between 4 and 9, and speeds up the naive method by a factor of 20 or more. Note that the absolute time required to compute the join is fairly modest—with $n = 30,000$, WHIRL takes well under a minute[13] to pick the best 10 answers from the 900 million possible candidates.

We also joined `ReutersTrain` and `Hoovers` using the company name column of `Hoovers` and the story column of `ReutersTrain`. This application of similarity joins corresponds to searching for all mentions in the `Reuters` corpus of any company listed in `Hoovers`, and illustrates an interesting blending of IR search with data integration. The results are shown in the second graph of Figure 2. On these problems the *maxscore* method does not improve over the naive method with respect to CPU time.[14] However, WHIRL speeds up the naive method by a factor of 2-4. The absolute time required is again small—about 5 CPU seconds for $n = 2474$.

It should be noted that the run-time for these queries is fast in part because some of the documents being joined are names. Names tend to be short and highly discriminative, and thus behave more like traditional database keys than arbitrary documents might. This point can be illustrated experimentally [9].

Elsewhere we present timing results for typical queries posed to a prototype data integration system based on WHIRL [10]. In this setting the queries are more complex (*e.g.*, four- and five-way joins) but the relations are somewhat smaller, containing a few hundred to a few thousand tuples. Query processing time for these queries is usually a tenth of a second or less.

4.2 Average precision on similarity joins

To evaluate the accuracy of the answers produced by WHIRL, we adopted the following methodology. Again focusing on similarity joins, we selected pairs of relations which contained two or more plausible "key" fields. One of these fields, the "primary key", was used in the similarity literal in the join. The second key field was then used to check the correctness of proposed pairings; specifically, a pairing was marked as "correct" if the secondary keys matched (using an appropriate matching procedure) and "incorrect" otherwise.

We then treated "correct" pairings in the same way that "relevant" documents are typically treated in evaluation of a ranking proposed by a standard IR system. In particular, we measured the quality of a ranking using *(non-interpolated)*

[12] In other experiment (not reported here due to space considerations) we have explored the result of increasing r up to several thousand. For these sorts of problems the compute time for WHIRL grows no worse than linearly with r.

[13] All timing results are given in CPU seconds on a MIPS Irix 6.3 with 200 MHz R10000 processors.

[14] It does, however, greatly reduce the number of accesses to the inverted index, as Turtle and Flood observed.

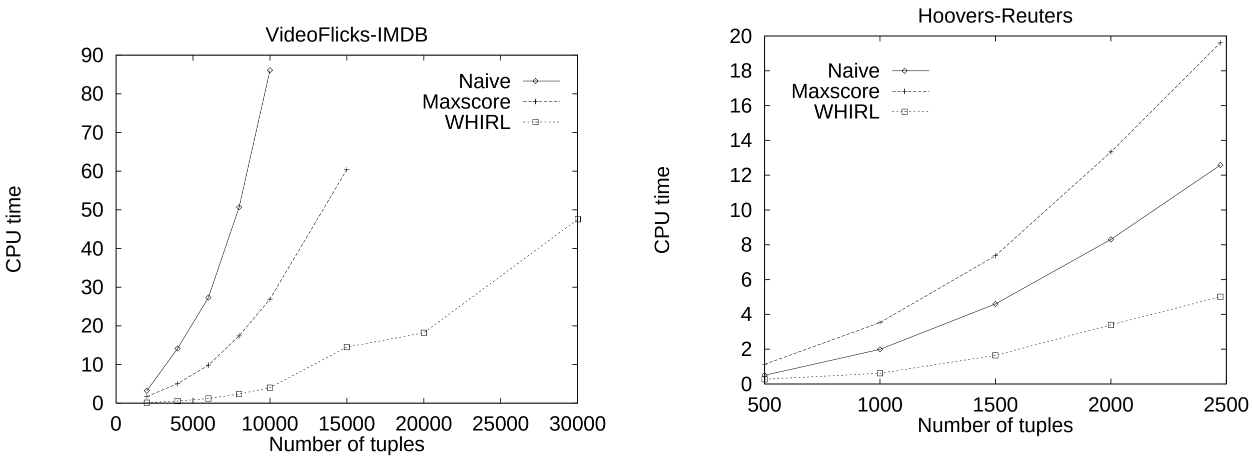

Figure 2: Runtime for similarity joins (in seconds)

Similarity Joins

Domain	Average Precision
Business	84.6%
Animals	92.1%
Movies	100.0%

Similarity Joins with Incompatible Schemata

Pairing	Average Precision
movieName/movieName	100.0%
movieListing/movieName	100.0%
movieName/review	98.0%
movieListing/review	94.0%

Table 2: Average precision for similarity joins

average precision. To motivate this measurement, assume the end user will scan down the list of answers and stop at some particular "target answer" that he or she finds to be of interest. The answers listed below this "target" are not relevant, since they are not examined by the user. Above the target, one would like to have a high density of correct pairings; specifically, one would like the set S of answers above the target to have high *precision*, where the precision of S is the ratio of the number of correct answers in S to the number of total answers in S. *Average precision* is the average precision for all "plausible" target answers, where an answer is considered a plausible target only if it is correct. To summarize, letting a_k be the number of correct answers in the first k, and letting $c(k) = 1$ iff the k-th answer is correct and letting $c(k) = 0$ otherwise, average precision is the quantity $\sum_{k=1}^{r} c(k) \cdot \frac{a_k}{k}$.

Note that average precision is 1 only when all correct answers precede all incorrect answers. In the experiments below, we used r-answers of size $r = 1000$ to compute average precision.

We used three pairs of relations from three different domains. In the business domain, we joined `Iontech` and `HooversWeb`, using company name as the primary key, and the string representing the "site" portion of the home page as a secondary key. In the movie domain, we joined `Review` and `MovieLink`, using film names as a primary key. As a secondary key, we used a special key constructed by the hand-coded normalization procedure for film names that is used in IM, an implemented heterogeneous data integration system [27]. In the animal domain, we joined `Animal1` and `Animal2`, using common names as the primary key, and scientific names as a secondary key (and a hand-coded domain-specific matching procedure).

The results are summarized in Table 2. On these domains, similarity joins are extremely accurate—in the movie domain, the performance is actually identical to the hand-coded normalization procedure. These results contrast with the typical performance of statistical IR systems on retrieval problems, where the average precision of a state-of-the art IR system is usually closer to 50% than 90%. This suggests that the similarity reasoning required to match names is easier than the similarity reasoning required to process a typical IR ranked retrieval query.

In the experiments, we used the secondary key as a "gold standard"; however, in most of the domains, the matching procedure for the secondary keys is somewhat error prone. Checking all pairings manually would be too time consuming, but to get some idea of the accuracy of the secondary keys we took the top 100 pairs in the business domain, and manually checked the 13 pairs marked as "incorrect" according to the secondary key. Of these 13 pairings, there were 11 in which the secondary keys were wrong, one in which the WHIRL pairing was wrong (at rank 77), and one pair where correctness could not be easily determined. This suggests that the similarity join is actually *more* accurate than the use of Web sites as a key.

4.3 Joins with incompatible schemata

Another problem which occurs in integrating heterogeneous data is the problem of incompatible schemata. For example, consider trying to find employees of a university using the following two relations: `professor(name, workAddress)` and `university(name,state)`. It is plausible that concatenating a university `name` and `state` would give a document similar, but not identical, to an employee's `workAddress`. (Typically a `workAddress` would have a number of extra terms, such as "Department of Computer Science", in addition to some variant of the university name and its state.) Thus in this case, an appropriate similarity join might give a useful result, even though the objects being joined are in fact different.

We explored this possibility by considering different schemata for the `MovieLink` and `Review` relations, with the aim of constructing problems that are similar to the sort of incompatible-schemata problem given above, but still possible to evaluate rigorously by checking individual pairings. For `MovieLink`, we considered a variation in which each tuple contains a single document containing a movie name plus a complete cinema address. (In the table, we call this document a "`movieListing`".) For `Review`, we considered a variation in which each tuple contains only a `review` entry, and no separate movie name field.[15] We then computed similarity joins with each possible combination of a `MovieLink` variant and a `Review` variant.

One would expect the irrelevant "noise" words that appear along with the movie names to have some adverse affect on precision. In our experiments with the `Review` and `Movielink` relations, however, the effect was quite slight: joining movie names to movie listings reduces average precision by only 2%, and joining movie listings to complete reviews reduces average precision by less than 7%. Finally, joining movie listings to movie names leads to no measurable loss in average precision. These results are summarized in Table 2.

5 Related Work

Chaudhuri *et al* present efficient solutions to the problem of loosely integrating Boolean text queries with database queries [8]. In contrast, we have considered a much tighter integration between databases and statistical IR queries. The assumptions made by Chaudhuri *et al* are not particularly appropriate in the context of heterogeneous database integration.

As noted above in Section 2.4, WHIRL is closely related to probabilistic databases (*e.g.*, [18; 4]). To our knowledge such database systems have not been used in data integration tasks. Furthermore, the implementation of WHIRL is unique in generating only a few "best" answers to a query; existing probabilistic database systems typically find all tuples with non-zero probability. As we argued above in Section 2.2, this would often be impractical for the problems encountered in this sort of heterogeneous database integration, due to the prevalence of weak matches between documents.

The WHIRL query algorithm borrows heavily from techniques previously used to optimize ranked retrieval searches in statistical IR. To our knowledge, these techniques have not been previously used for approximating the join of lists of documents. More generally, the sort of approximate join implemented in WHIRL does not seem to have been investigated in the IR literature, although numerous other hybrids of statistical IR techniques with database representations have been proposed (*e.g.*, [37; 18]).

There has also been much work on approximate matching techniques for the removal of duplicates and merging of heterogeneous data sources [32; 16; 22; 21; 20; 31]. Most of the approximate matching methods proposed are domain-specific (*e.g.*, using Soundex to match surnames), a notable exception being the Smith-Waterman edit distance adopted by Monge and Elkan [31]. Applying these techniques is a relatively expensive off-line process which is usually not guaranteed to find the best matches, due to the nearly universal use of "blocking" heuristics which restrict the number of similarity comparisons.

Here, we have considered approximate matching using the vector space model of similarity. This model enjoys a number of advantages. Like Smith-Waterman, it is domain-independent. It is extremely well supported experimentally as a similarity metric for text; we note that in a previous comparison, a simple term-weighting method gave better matches than the Smith-Waterman metric [30]. Finally, by using inverted indices, it is possible to quickly locate items similar to a given item. Exploitation of this property results in a approximate matching algorithm that is guaranteed to find the best pairings, but still fast enough to interleave with query-answering. Note that interleaving matching with query-answering, rather than computing the best matches off-line, has an important consequence: rather than commit early as to whether a match is correct or incorrect, one can propagate uncertainty about approximate matches, and then use the propagated uncertainty to rank answers presented to the end user.

There have also been a number of approaches to data integration which address issues orthogonal to the problem of lack of common domains. Examples of such work include "semi-structured" data models [38; 1; 39]; while we have focused here on relational models, due to their simplicity, we believe that many of the basic principles of WHIRL can be applied to more complex data models as well. Other data integration systems provide a database-like view of the Web (*e.g.*, [17; 29; 24]), in which queries can express combinations of keyword searches and hypertext connectivity constraints; in effect, these languages offer a means declaratively navigating the Web. As suggested by the experiments, we believe that our work is most appropriate for integrating sites that contain no explicit links connecting them. WHIRL also differs from such models in that it includes statistical IR methods for searching within documents, rather than boolean keyword search methods.

In its basic motivation, our work is inspired by previous work in the integration of heterogeneous data sources, such as data sources on the Web [27; 2; 19; 3; 40; 6]. None of these previous systems, however, include a "fuzzy" matching procedure for names; instead they generally construct global domains using hand-crafted domain-specific normalization schemes. Use of domain-specific matching algorithms has also been proposed as an alternative to normalization [15].

The connection between WHIRL and other data integration systems is discussed more fully in another paper [10], which describes a WHIRL-based data integration system for Web data sources. The focus of that paper is on mechanisms for converting HTML information sources into STIR databases, and other practical issues in fielding a data integration system. In contrast, this paper focuses on efficient theorem proving algorithms for WHIRL and rigorous evaluation of WHIRL in controlled experiments.

Some of the results of this paper have also appeared previously in preliminary form [9].

6 Conclusions

In an ideal world, one would like to integrate information from heterogeneous autonomous databases with little or no human effort. In other words, one would like data to be *easily shared among databases*. Unfortunately, such data sharing is difficult with current data models. One fundamental and critical problem is the *lack of global domains:* different databases are likely to use different constants to refer to the same real-world entity, making operations like joins across relations from different databases impossible.

We believe the data model and query language presented in this paper represent a significant advance toward the long-term goal of easily sharable data. We have outlined an approach to the integration of structured heterogeneous infor-

[15] The `review` documents virtually always contain a title naming the movie being reviewed, as well as a lot of additional text.

mation sources, based on extended conventional database query languages with standard IR methods for reasoning about textual similarity. The approach is embodied in an implemented logic called WHIRL. WHIRL is intended for integration of relations that are semantically heterogeneous in the sense that there is no common naming scheme for entities.

The problem of integrating relations without global domains has received little prior attention. Current data integration systems typically use domain-specific rules to normalize entity names, and then use the normalized versions of these names as keys. These normalization rules are developed manually, sometimes at considerable effort. In practice, the cost of this process in terms of human time limits data integration systems to relatively well-structured data collected from a relatively small number of sites. Furthermore, normalization is prone to error, and unlike WHIRL, a system based on normalized keys has no way of either assessing the likelihood of such errors or (more importantly) informing the user of potential errors.

Our experiments show that the accuracy of WHIRL's "similarity joins" are quite good, even compared to hand-coded integration schemes based on normalization. In one case WHIRL's performance equals the performance of a hand-constructed, domain-specific normalization routine. In a second case, WHIRL's performance gives better performance than matching on a plausible global domain. WHIRL is also efficient; the current implementation can handle join operations on moderate sized databases (containing a few tens of thousands of tuples) at interactive speeds.

Although these results are encouraging, many additional topics remain to be addressed. There are many well-known methods for conducting an approximate A$*$ search; some or all of these may lead to substantial performance improvements. The current version of WHIRL handles heterogeneous data, but not in a distributed fashion; this is another intriguing topic for future work. We would also like to consider the issue of closely integrating WHIRL with appropriate learning methods for text categorization [28; 12], adjusting numerical parameters for queries [5; 7; 11], and learning logical expressions [35].

Finally, we plan to continue our evaluation of WHIRL on actual data integration tasks [10]. These experiments allow us to evaluate WHIRL on less artificial queries, and suggest necessary extensions. One drawback of such work, however, is that any system integrating data from *existing* Web information sources requires some sort of human-directed translation of these sources—for instance, our data integration system requires translation from HTML to STIR. This makes it impossible to separate the performance the integration system as a whole from the cleverness and industry of the humans that are doing the translation. In a better world, of course, translation would be unnecessary; instead data would be encoded directly in STIR, or some other sharable data model.

Acknowledgments

The author is grateful to Alon Levy for numerous helpful discussions while I was formulating this problem, and for comments on a draft of the paper; to Jaewoo Kang, for providing me with data and the normalization routines used in IM; to Alex Borgida, Sal Stolfo, and Mark Jones for comments on the paper; to Susan Cohen for proofreading; and to Edith Cohen, David Lewis, Haym Hirsh, Fernando Pereira, Divesh Srinivasan, Dan Suciu, and many other colleagues for helpful advice and discussions.

References

[1] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-97)*, Tucson, AZ, May 1997.

[2] Yigal Arens, Craig A. Knoblock, and Chun-Nan Hsu. Query processing in the SIMS information mediator. In Austin Tate, editor, *Advanced Planning Technology*. AAAI Press, Menlo Park, CA, 1996.

[3] Paolo Atzeni, Giansalvatore Mecca, and Paolo Merialdo. Semistructured and structured data on the Web: going back and forth. In Dan Suciu, editor, *Proceedings of the Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997. Available on-line from http://www.research.att.com/ suciu/workshop-papers.html.

[4] Daniel Barbara, Hector Garcia-Molina, and Daryl Porter. The management of probabilistic data. *IEEE Transations on knowledge and data engineering*, 4(5):487–501, October 1992.

[5] Brian T. Bartell, Garrison W. Cottrell, and Richard K. Belew. Automatic combination of multiple ranked retrieval systems. In *Seventeenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1994.

[6] R. J. Bayardo, W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishan, A. Unruh, and D. Woelk. Infosleuth: an agent-based semantic integration of information in open and dynamic environments. In *Proceedings of the 1997 ACM SIGMOD*, May 1997.

[7] Justin Boyan, Dane Freitag, and Thorsten Joachims. A machine learning architecture for optimizing web search engines. Technical Report WS-96-05, American Association of Artificial Intelligence, 1994.

[8] S. Chaudhuri, U. Dayal, and T. Yan. Join queries with external text sources: execution and optimization techniques. In *Proceedings of the 1995 ACM SIGMOD*, May 1995.

[9] William W. Cohen. Knowledge integration for structured information sources containing text (extended abstract). In *The SIGIR-97 Workshop on Networked Information Retrieval*, 1997.

[10] William W. Cohen. A Web-based information system that reasons with structured collections of text. In *Proceedings of Autonomous Agents-98*, St. Paul, MN, 1998.

[11] William W. Cohen, Rob Schapire, and Yoram Singer. Learning to order things. To appear in NIPS-97, 1997.

[12] William W. Cohen and Yoram Singer. Context-sensitive learning methods for text categorization. In *Proceedings of the 19th Annual International ACM Conference on Research and Development in Information Retrieval*, pages 307–315, Zurich, Switzerland, 1996. ACM Press.

[13] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-97)*, Tucson, AZ, May 1997.

[14] Oliver M. Duschka and Michael R. Genesereth. Query planning in infomaster. In *Proceedings of the Twelfth Annual ACM Symposium on Applied Computing (SAC97)*, San Jose, CA, February 1997.

[15] Douglas Fang, Joachim Hammer, and Dennis McLeod. The identification and resolution of semantic heterogeneity in multidatabase systems. In *Multidatabase Systems: An Advanced Solution for Global Information Sharing*, pages 52–60. IEEE Computer Society Press, Los Alamitos, California, 1994.

[16] I. P. Felligi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Society*, 64:1183–1210, 1969.

[17] Thorsten Fiebig, Jurgen Weiss, and Guido Moerkotte. RAW: a relational algebra for the Web. In Dan Suciu, editor, *Proceedings of the Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997. Available on-line from http://www.research.att.com/ suciu/workshop-papers.html.

[18] Norbert Fuhr. Probabilistic Datalog—a logic for powerful retrieval methods. In *Proceedings of the 1995 ACM SIGIR conference on research in information retrieval*, pages 282–290, New York, 1995.

[19] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS approach to mediation: Data models and languages (extended abstract). In *Next Generation Information Technologies and Systems (NGITS-95)*, Naharia, Israel, November 1995.

[20] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD*, May 1995.

[21] Scott Huffman and David Steier. Heuristic joins to integrate structured heterogeneous data. In *Working notes of the AAAI spring symposium on information gathering in heterogeneous distributed environments*, Palo Alto, CA, March 1995. AAAI Press.

[22] B. Kilss and W. Alvey (ed). Record linkage techniques—1985. Statistics of Income Division, Internal Revenue Service Publication 1299-2-96. Available from http://www.bts.gov/fcsm/methodology/, 1985.

[23] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms (second edition)*. Addison-Wesley, Reading MA, 1975.

[24] D. Konopnicki and O. Schmueli. W3QS: a query system for the world wide web. In *Proceedings of the 21nd International Conference on Very Large Databases (VLDB-96)*, Zurich, Switzerland, 1995.

[25] Richard Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, July 1993.

[26] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Query answering algorithms for information agents. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, Portland, Oregon, august 1996.

[27] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB-96)*, Bombay, India, September 1996.

[28] David Lewis. Representation and learning in information retrieval. Technical Report 91-93, Computer Science Dept., University of Massachusetts at Amherst, 1992. PhD Thesis.

[29] Alberto Mendelzon and Tovo Milo. Formal models of Web queries. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-97)*, Tucson, AZ, May 1997.

[30] A. Monge and C. Elkan. The field-matching problem: algorithm and applications. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, August 1996.

[31] A. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *The proceedings of the SIGMOD 1997 workshop on data mining and knowledge discovery*, May 1997.

[32] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130:954–959, 1959.

[33] Nils Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1987.

[34] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[35] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3), 1990.

[36] Gerard Salton, editor. *Automatic Text Processing*. Addison Welsley, Reading, Massachusetts, 1989.

[37] Peter Schäuble. SPIDER: A multiuser information retrieval system for semistructured and dynamic data. In *Proceedings of the 1993 ACM SIGIR conference on research in information retrieval*, pages 318–327, Pittsburgh, PA, 1993.

[38] Dan Suciu. Query decomposition and view maintanance for query languages for unstructured data. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB-96)*, Bombay, India, 1996.

[39] Dan Suciu, editor. *Proceedings of the Workshop on Management of Semistructured Data*. Available on-line from http://www.research.att.com/ suciu/workshop-papers.html, Tucson, Arizona, May 1997.

[40] Anthony Tomasic, Remy Amouroux, Philippe Bonnet, and Olga Kapitskaia. The distributed information search component (Disco) and the World Wide Web. In *Proceedings of the 1997 ACM SIGMOD*, May 1997.

[41] Howard Turtle and James Flood. Query evaluation: strategies and optimizations. *Information processing and management*, 31(6):831–850, November 1995.