# TOWARD ENTITY-AWARE SEARCH

BY

TAO CHENG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Professor Jiawei Han, Chair
Associate Professor Kevin Chang, Director of Research
Associate Professor ChengXiang Zhai
Doctor Gerhard Weikum, Max Planck Institute

# Abstract

As the Web has evolved into a data-rich repository, with the standard "page view," current search engines are becoming increasingly inadequate for a wide range of query tasks. While we often search for various data "entities" (*e.g.*, phone number, paper PDF, date), today's engines only take us indirectly to pages. In my Ph.D. study, we focus on a novel type of Web search that is aware of data entities inside pages, a significant departure from traditional document retrieval. We study the various essential aspects of supporting entity-aware Web search. To begin with, we tackle the core challenge of ranking entities, by distilling its underlying conceptual model *Impression Model* and developing a probabilistic ranking framework, *EntityRank*, that is able to seamlessly integrate both local and global information in ranking. We also report a prototype system built to show the initial promise of the proposal. Then, we aim at distilling and abstracting the essential computation requirements of entity search. From the dual views of reasoning–*entity as input* and *entity as output*, we propose a dual-inversion framework, with two indexing and partition schemes, towards efficient and scalable query processing. Further, to recognize more entity instances, we study the problem of entity synonym discovery through mining query log data. The results we obtained so far have shown clear promise of entity-aware search, in its usefulness, effectiveness, efficiency and scalability.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

The immense scale and wide spread of the Web has rendered it an ultimate information repository– It contains all kinds of *data,* much beyond the conventional *page view* of the Web as a corpus of "documents." While the richness of data represents a promising opportunity, it challenges us for effectively finding information we need, beyond the traditional search paradigm of retrieving relevant documents.

On the other side, we have also noticed tremendous needs for *data entities* from the end users. Two recent studies have revealed that a significant portion of all Web queries are about entities. Specifically, Kumar et al. show that 52.9% of web search queries are entity-oriented queries [47]. And Guo et al. show that about 71% of web search queries contain named entities [37].

Search today is mostly oblivious to *data* embedded in various pages– Can we build the awareness of entities into large scale search over the Web to bridge the users' need of entities with the rich set of data entities on the Web?

Toward *entity-aware* search, to focus on the "stuff" we want, or data "entities,", we have been developing the concept and system for *entity search*, in the WISDM project (http://wisdm.cs.uiuc.edu) at the University of Illinois. In entity search, the Web is conceptualized as a repository of data *entities*, which appear in various linked documents. As Figure 1.1 shows, while traditional search views the Web as a collection of pages, and finds pages by keywords, an entity-search system views the Web as a repository of (various types of) entities, and searches these entities directly.

Traditional search paradigms view the Web as a document collection $\mathcal{D}$ of documents $\{d_1, d_2, \ldots, d_n\}$. We take an *entity view* of the Web: we consider the Web as primarily a repository of entities (in addition to the notions of pages): $E = \{E_1, E_2, \ldots, E_N\}$, where each $E_i$ is an entity type. (*e.g.*, $E = \{E_1 : \#\mathsf{phone}, E_2 : \#\mathsf{email}\}$). We use a prefix # sign (*e.g.*, #phone for phone entity) throughout the thesis to distinguish entities from keywords. Further, each entity type $E_i$ is a set of *entity instances* that are extracted from the corpus, *i.e.*, literal values of entity type $E_i$ that occur somewhere in some document $d \in \mathcal{D}$. We use $e_i$ to denote an entity instance of entity type $E_i$. In the example of phone-number patterns, we may extract #phone = {"800-2017575", "244-2919", ...}

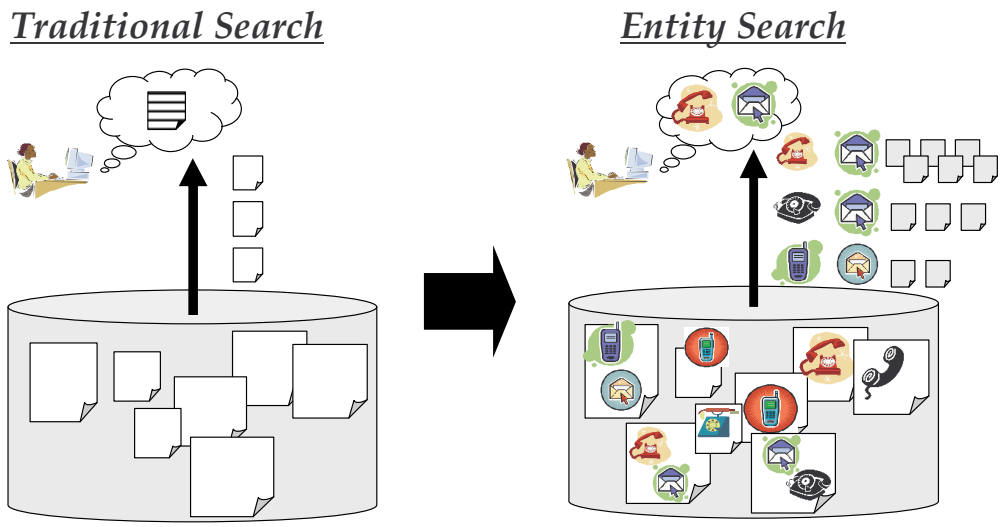



Figure 1.1: From Tradition Search to Entity Search

To motivate concretely, consider user Amy: She may be looking for the "phone number" of say, Amazon.com's customer service? When preparing seminar presentation, Amy wants to find papers that come readily with presentations, *i.e.*, a "PDF file" *together with* a "PPT file," say from SIGMOD 2006? In these scenarios, Amy is looking for particular entities of information, *e.g.*, a phone number, a PDF, a PPT, *etc.*. To illustrate, our scenarios will lead to the following queries:

$Q1$: ow20(amazon service #phone)

$Q2$: uw (sigmod 2006 #pdf_file #ppt_file)

As *input*, users formulate queries to directly describe what they are looking for: She can simply specify what her *target entities* are and what keywords may appear in the surrounding context with a right answer. Each query is thus a *context pattern* of how the desired entity may occur with some keywords in its surrounding context. $Q1$ says that the entity #phone will appear with these keywords in the pattern of ow20 or "ordered-window of 20 words" (and as close as possible).

As *output*, users directly get their desired entities. That is, as a query specifies the target entity types, its results are those entity *instances* (or literal values) that match the query, in a ranked order by matching scores. Figure 2.2 shows example results for $Q1$ and $Q2$. Here pages, unlike being the primary search target in document search, become the supporting evidence for the entity results.

| rank | phone number | score | urls |
|---|---|---|---|
| 1 | 800-201-7575 | 0.9 | amazon.com/support.htm<br>myblog.org/shopping |
| 2 | 800-988-0886 | 0.8 | Dell.com/supportors |
| 3 | 800-342-5283 | 0.6 | xyz.com |
| 4 | 206-346-2992 | 0.2 | hp.com |
| ... | ... | ... | ... |

| rank | PDF | PPT | score | urls |
|---|---|---|---|---|
| 1 | sigmod6.pdf | sigmod6.ppt | 0.8 | db.com,sigmod.com |
| 2 | surajit21.pdf | surajit21.ppt | 0.7 | ms.com |
| ... | ... | ... | ... | ... |

Figure 1.2: Query Results of $Q1$ and $Q2$

## 1.1 Entity Search Problem Definition

We proposed the concept of "entity search" and formally defined the problem in [21]. We now introduce the underlying data model we use, and present the definition of the entity search problem.

### 1.1.1 Data Model: Entity View

How should we view the Web as our database to search over? In the standard page view, the Web is a set of documents (or pages) $\mathcal{D} = \{d_1, \ldots, d_n\}$. We assume flat set for discussion here; Chapter 2 will specialize $\mathcal{D}$ as a set of linked documents.

In our data model, we take an *entity view*: We consider the Web as primarily a repository of entities (in addition to the notions of pages): $E = \{E_1, E_2, \ldots, E_n\}$, where each $E_i$ is an entity type. For instance, to support query $Q1$ as we motivated earlier in this chapter, the system might be constructed with entities $E = \{E_1 : \#\mathsf{phone}, E_2 : \#\mathsf{email}\}$. Further, each entity type $E_i$ is a set of *entity instances* that are extracted from the corpus, *i.e.*, literal values of entity type $E_i$ that occur somewhere in some $d \in \mathcal{D}$. We use $e_i$ to denote an entity instance of entity type $E_i$. In the example of phone-number patterns, we may extract $\#\mathsf{phone} = \{$"800-201-7575", "244-2919", "(217) 344-9788, ...$\}$

In this work, we consider only entities that can be recognized offline before query-time, and the extracted instances will be indexed for efficient query processing. The extraction can be done using simple pattern

***Entity-Search* Query.**

- **Given**: *Entity collection* $\mathcal{E} = \{E_1, \ldots, E_N\}$, over *Document collection* $\mathcal{D} = \{d_1, \ldots, d_n\}$.
- **Input**: *Query* $q(\langle E_1, \ldots, E_m \rangle) = \alpha(E_1, \ldots, E_m, k_1, \ldots, k_l)$, where $\alpha$ is a *tuple pattern*, $E_i \in \mathcal{E}$, and $k_j$ a keyword.
- **Output**: *Ranked list* of $t = \langle e_1, \ldots, e_m \rangle$, where $e_i \in E_i$, sorted by $Score(q(t))$, the *query score* of $t$.

Figure 1.3: The Entity Search Problem

matching or state-of-the-art entity extractors.

To facilitate query matching, we will record the "features" of each occurrence $e_i$.

- Position $e_i.pos$: the document id and word offset of this instance occurrence, *e.g.*, *instance* $e_1$ may occur at ($d_2$, 23).
- Confidence $e_i.conf$: the probability estimation that indicates how this occurrence is regarded as an instance of $E_i$.

With entities extracted and indexed, we transform the page view into our entity view. Note that the set of supported entity types must be determined, depending on the actual application setting, much like the "schema" of the system.

We stress that each of these entities are *independently* extracted from the corpus, and are only associated by ad-hoc queries at query time. Thus, users may ask #phone with "ibm thinkpad" or "bill gates", or they ask to pair #phone with, say, #email for "white house". Supporting such online matching and association is exactly the challenge (and usefulness) of entity search.

### 1.1.2 Search Problem: Finding Entity Instances

We now state our entity search problem, as Figure 1.3 summarizes. First, for *input*, as queries, our entity search system lets users search for entities by specifying target entity types and keywords together in a *tuple pattern* $\alpha$, which indicates users' intention of what the desired entities are, and how they may appear in $\mathcal{D}$ by certain patterns. We note that entity search is essentially *search by context* over the document collection: As $\alpha$ intends to capture, our desired data often appear in some context patterns with other keywords or entities, indicating how they together combine into a desired *tuple* by their textual occurrences.

A system will support, as its implementation decisions, a set of such patterns, *e.g.*, doc (the same document), ow (ordered window), uw (unordered window), and phrase (exact matching). A query can either explicitly specify a pattern (*e.g.*, $Q1$) or implicitly assume the system default pattern (*e.g.*, $Q2$).

Second, for *output*, the results are a ranked list of $m$-ary *entity tuples*, each of the form $t = \langle e_1, \ldots, e_m \rangle$, *i.e.*, a combined instance of each $e_i$ as an instance of entity $E_i$ desired in the query. A result tuple $t$ will be ranked higher, if it matches the query better. We denote this measure of how well $t$ matches $q$ by a *query score* $Score(q(t))$, which should capture how $t$ appears, by the desired tuple pattern $\alpha$, across every document $d_j$ in $\mathcal{D}$, *i.e.*,

$$Score(q(t)) = Score(\alpha(e_1, \ldots, e_m, k_1, \ldots, k_l)).$$

We stress that, since the assessment of the query score defines the ranked list, it is the central function of an entity search system. The objective of entity search is thus to find from the space of $t \in E_1 \cdot \ldots \cdot E_m$, the matching tuples in ranked order by how well they match $q$, *i.e.*, how well entity instances and keywords in tuple $t$ associate in the desired tuple pattern.

## 1.2 Our Contributions

We summarize the contributions of our study as follows:

Problem: We are among one of the first to propose, formulate and study the entity search problem [21].

System: A system prototype built over real Web corpus [25] to concretely validate our ideas and designs.

Techniques: Systematic study of the search requirements and the design of a conceptual Impression model and a concrete EntityRank ranking model [24] for entity retrieval effectiveness. Principled study of index design, partition schemes and query processing to enable efficient and scalable entity search [22] at large scale. Study of the entity synonym discovery problem for enabling entity resolution [23].

The rest of the thesis is organized as follows:

Chapter 2 focuses on search effectiveness by studying ranking models. With an effective ranking model established, we next study Chapter 3 indexing design and parallelization schemes for high search efficiency and scalability. We also study the entity synonym discovery problem in Chapter 4, to reveal the various representations of an entity in the form of entity synonyms. Chapter 5 reviews the related systems and Chapter 6 concludes the thesis.

# Chapter 2

# Entity Ranking: Searching Entities Directly and Holistically

## 2.1 Introduction

The immense scale and wide spread of the Web has rendered it as an ultimate information repository– as not only the sources where we *find* but also the destinations where we *publish* our information. These dual forces have enriched the Web with all kinds of *data,* much beyond the conventional *page view* of the Web as a corpus of HTML pages, or "documents." Consequently, the Web is now a collection of *data-rich* pages, on the "surface Web" of static URLs (*e.g.*, personal homepages) as well as the "deep Web" of database-backed contents (*e.g.*, flights from aa.com), as Figure 2.1 shows. While the richness of data represents a promising opportunity, it challenges us for effectively finding information we need.

With the Web's sheer size, our ability to find "stuff" we want mainly relies on how *search engines* respond to our *queries*. As current engines search the Web inherently with the conventional page view, they are becoming increasingly inadequate for a wide range of queries. To focus on the "stuff" we want, or data "entities", this chapter studies the *entity search* problem, formulates the search framework, and in particular addresses the central issue of *entity ranking*.

### Motivating Scenarios: The Barriers

To begin with, we reflect: As users, what have we been looking for on the Web? The richness of data has tempted us to search for various "stuff"– Let us consider a few scenarios, for user Amy:

**Scenario 1:** Amy wants to call Amazon.com for her online purchase; how can she find the "phone number" of their customer service? To begin with, what should be the right keywords for finding pages with such numbers? Query "amazon customer service phone" may not work, as often a phone is simply shown without keyword "phone" (*e.g.*, customer service: (800) 717-6688). Or, "amazon customer service" could be too broad to return many pages. Amy must sift through the returned pages to dig for the phone number. This task can be time consuming, since some vendors may "hide" their service numbers (to reduce their workload). Fortunately, such information might reside in other probably less authoritative (or lower ranked)

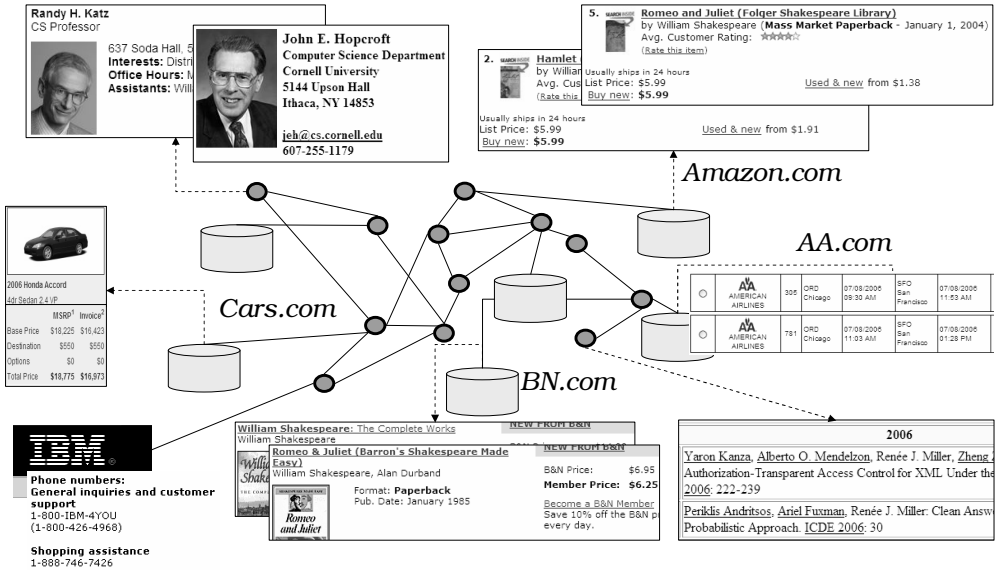



Figure 2.1: The Current Web: Proliferation of Data-Rich Pages

pages, *e.g.*, user forums, business reviews, and blogs. □

**Scenario 2:** Amy wants to apply for graduate schools; how can she find the *list* of "professors" in the database area? She might have to go through all the CS department homepages (in the hope that there is a page that lists professors by research areas), or look through faculty homepages one by one. This could be a very laborious process. □

**Scenario 3:** As a graduate student, Amy needs to prepare a seminar presentation for her choice of some recent papers; how can she find papers that come readily with presentations, *i.e.*, a "PDF file" *together with* a "PPT file," say from SIGMOD 2006? □

**Scenario 4:** Done with the presentation, now Amy wants to buy a copy of Shakespeare's Hamlet to read; how can she find the "prices" and "cover images" of available choices from, say, Borders.com and BN.com (She has seen the copy she likes before, so the cover page will be helpful to find out). She would have to look at the results from multiple online bookstores one by one and compare the listed price of each. □

In these scenarios, like every user in many similar situations, Amy is looking for particular *types* of information, which we call *entities*, *e.g.*, a phone number, a book cover image, a PDF, a PPT, a name, a date, an email address, etc. She is *not*, we stress, looking for pages as "relevant documents" to read, but entities as data for her subsequent tasks (to contact, to apply, to present, to buy, *etc.*).

However, the current search systems, with their inherent page view of the Web, are inadequate for the task of finding data entities, the focus of this chapter. There are two major barriers:

*First*, in terms of the *input and output*, current engines are searching *indirectly*. 1) Users cannot directly describe what they want. Amy has to formulate her needs indirectly as keyword queries, often in a non-trivial and non-intuitive way, with a hope to hit "relevant pages" that may or may not contain target entities. For Scenario 1, will "amazon customer service phone" work? (A good page may simply list the phone number, without the word "phone.") 2) Users cannot directly get what they want. The engine will only take Amy to a list of pages, and she must scrutinize them to find the phone number. Can we help Amy to *search directly* in both describing and getting what they want?

*Second*, in terms of the *matching mechanism*, current search engines are finding each page *individually*. The target entities are often available in multiple pages. In Scenario 1, the same phone number of Amazon.com may appear in the company's Web site, online user forums, or even blogs. In this case, we should collect, for each phone, all its occurrences from multiple pages as supporting evidences of matching. In Scenario 2, the list of professors probably cannot be found in any single page. In this case, again, we must look at many pages to come up with the list of promising names (and similar to Scenario 1, each name may appear in multiple pages). Can we help users to *search holistically* for matching entities across the Web corpus as a whole, instead of individual pages?

## Our Proposal: Entity Search

Toward searching directly and holistically, for finding specific types of information, we propose to support *entity search*.

First, as *input*, users formulate queries to directly describe what they are looking for: She can simply specify what her *target entities* are, and what keywords may appear in the *context* with a right answer. To distinguish between entities to look for and keywords in the context, we use a prefix #, *e.g.*, #phone for the phone entity. Our scenarios will naturally lead to the following queries:

**Query** $Q1$:ow(amazon customer service #phone)

**Query** $Q2$: (#professor #university #research="database")

**Query** $Q3$:ow(sigmod 2006 #pdf_file #ppt_file)

**Query** $Q4$: (#title="hamlet" #image #price)

In these queries, there are two components: (1) *Context* pattern, how will the target entities appear? $Q1$ says that the entity #phone will appear with these keywords in the pattern of ow or "ordered-window", *i.e.*, in that order and as close in a window as possible. We may also omit the pattern, *e.g.*, $Q2$ and $Q4$, in which case the implicit default uw or "unordered-window" is used (which means proximity– the closer in a window, the better). (The exact patterns depend on implementations. Section 2.4 will discuss the notion of

| rank | phone number | score | urls |
|---|---|---|---|
| 1 | 800-201-7575 | 0.9 | amazon.com/support.htm<br>myblog.org/shopping |
| 2 | 800-988-0886 | 0.8 | Dell.com/supportors |
| 3 | 800-342-5283 | 0.6 | xyz.com |
| 4 | 206-346-2992 | 0.2 | hp.com |
| ... | ... | ... | ... |

| rank | PDF | PPT | score | urls |
|---|---|---|---|---|
| 1 | sigmod6.pdf | sigmod6.ppt | 0.8 | db.com,sigmod.com |
| 2 | surajit21.pdf | surajit21.ppt | 0.7 | ms.com |
| ... | ... | ... | ... | ... |

Figure 2.2: Query Results of $Q1$ and $Q3$

such "recognition models.") (2) *Content* restriction: A target entity will match any instances of that entity type, subject to option restriction on their content values– *e.g.*, $Q1$ will match every phone instance, while $Q2$ will only match research area "database." (In addition to equality "=", other restriction operators are possible, such as "contain.")

Second, as *output*, users will directly get the entities that they are looking for. That is, as a query specifies what entity types are the targets, its results are those entity *instances* (or literal values) that match the query, in a ranked order by their matching scores. (We will discuss this matching next.) Figure 2.2 shows some example results for $Q1$ and $Q3$.

Third, as *search mechanism*, entity search will find matching entities holistically, where an instance will be found and matched in all the pages where it occurs. For instance, a #phone 800-201-7575 may occur at multiple URLs as Figure 2.2 shows. For each instance, all its matching occurrences will be aggregated to form the final ranking– *e.g.*, a phone number occurs more *frequently* at where "amazon customer service" is mentioned may rank higher than those less frequent ones. (This ranking is our focus– See next.) Thus, while our search target is entities, as supporting "evidences," entity search will also return where each entity is found. Users can examine these snippets for details.

We note that, the usefulness of entity search is three-fold, as the sample results in Figure 2.2 illustrate. *First*, it returns relevant answers at top rank places, greatly saving search time and allowing users or applications to focus on top results. *Second*, it collects all the evidences regarding the query in the form of listing supporting pages for every answer, enabling results validation (by users) or program-based post-processing (by applications). *Third*, by targeting at typed entities, such an engine is data-aware and can be

integrated with DBMS for building novel information systems– imagine the results of $Q1$ to $Q4$ are connected with SQL-based data.

## Core Challenge: Ranking Entities

Toward building an entity search engine, we believe the core challenge lies in the entity ranking model– Obviously, while promising, such a system is only useful if good entity results can be found at the top ranks, much like today's search engines that strive to achieve the central mission of ranking relevant pages high.

As our discussion has hinted, there are several unique requirements of entity search. Entity search is 1) *contextual*, as it is mainly matching by the surrounding context; 2) *holistic*, entities must be matched across their multiple occurrences over different pages; 3) *uncertain*, since entity extraction is imperfect in nature; 4) *associative*, entities can be associated in pairs, *e.g.*, #phone and #email and it is important to tell true association from accidental; and 5) *discriminative*, as entities can come from different pages, and not all such "sources" are equivalent.

With these requirements, this chapter focuses on entity ranking: We build our foundation by proposing the *impression model*, an "ideal" conceptual framework. With the conceptual guidance, we build the *EntityRank* scheme, taking a principled probabilistic view for scoring and ranking: We conceptualize the matching of a result as to estimate the *probability* how the entity instances are associated as a *tuple*, and compare it to a *null hypothesis* to discriminate accidental associations. With a *local recognition layer* for quantifying each instance occurrence, a *global access layer* for aggregating across pages, and a *validation layer* for hypothesis testing, *EntityRank* materializes the conceptual impression model.

Our results show that *EntityRank* is effective: In our prototype indexing 2 TB of real Web corpus, for Scenario 1, it consistently finds the right matches at top-3, for a sample of Fortune 500 companies, and similarly for a systematic querying of SIGMOD 2007 PC members. Section 2.5 will demonstrate the results for all four scenarios. We validate the seamless integration of local recognition and global access models– without either, the results are significantly degraded– as well as the need for hypothesis testing.

We start in Section 2.2 to formalize entity search. Section 2.3 presents the ideal conceptual model and Section 2.4 materializes it into the *EntityRank* scheme. We relate to existing studies in Section 4.5, and Section 2.5 reports our prototype system and experiments.

## Contributions

1. We study and define the **characteristics and requirements** of entity search as the guideline for supporting effective ranking.

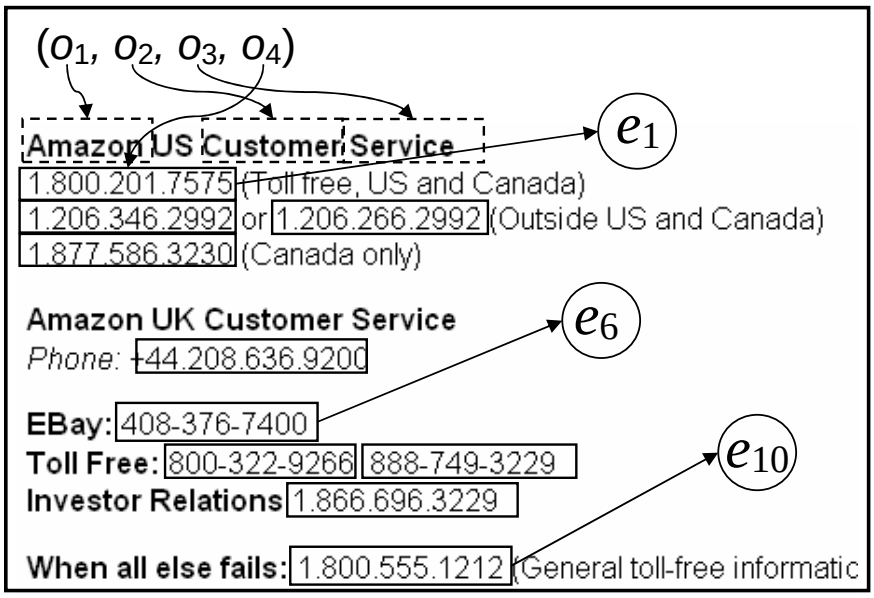



Figure 2.3: Example Page: Amazon Customer Service #phone

2. We distill the conceptual model **Impression Model**, and develop a concrete **EntityRank** framework for ranking entities.
3. We have implemented an online **prototype** with **real Web corpus**, and demonstrated the effectiveness of entity search.

## 2.2 Problem and Requirements

To support entity-based querying, the system must be fundamentally *entity-aware*: That is, while current search engines are built around the notion of pages and keywords, we must generalize them to support entity as a first-class concept. With this awareness, as our data model, we will move from the current page view, *i.e.*, the Web as a document collection, to the new *entity view*, *i.e.*, the Web as an entity repository. Upon this foundation, we develop entity search, where users specify what they are looking for with keywords and target entities, as $Q1$ – $Q4$ illustrated.

### 2.2.1 Entity Ranking: Requirements

For effective entity ranking, it is crucial to capture the unique characteristics of entity search. Let's examine a sample query: find the phone number of "Amazon Customer Service"; $q$ = (Amazon Customer Service #phone).

For each query, conceptually, an entity search system should analyze all pages available on the Web that

contain the keywords "Amazon Customer Service" and a #phone instance. Figure 2.3 is a text snippet from an example webpage that contains keywords "Amazon Customer Service" and #phone instances. There could be many such pages. The first step, for such a page, is to match the keywords "Amazon Customer Service" and identify the entity instances desired (#phone). Next, we must rank these entity instances since there might be multiple phone numbers co-located with "Amazon Customer Service" in webpages. The ranking function eventually will single out which phone number is associated with "Amazon Customer Service" more strongly. An entity search system needs to take into account the following major factors (as Section 3.1 briefly mentioned).

- R-*Contextual*: The probability of association between keywords and entity instances in various contexts might be different. There are mainly two factors to consider:
  a) Pattern: The association of keywords and entity instances sometimes formulates a regular pattern; *e.g.*, a company's name often appears *before* its phone number is mentioned. Given a text snippet "Amazon 1-800-201-7575 eBay," the phone number is more likely to associate with "Amazon" than "eBay".
  b) Proximity: The association between the keywords and the entity instances is not equally probable with respect to how "tightly" they appear in the web page. Often, the association is stronger when the occurrences are closer. Use Figure 2.3 as an example. Phone number $e_1$ 1-800-201-7575 is more likely associated with Amazon than phone number $e_6$ 408-376-7400 as $e_1$ appears in closer proximity to keywords "Amazon Customer Service" than $e_6$.
- R-*Holistic*: As a specific phone number instance may occur with "Amazon Customer Service" multiple times in many pages, all such matchings must be aggregated for estimating their association probability.
- R-*Uncertainty*: Entity extraction is always not perfect, and its extraction confidence probability must be captured.
- R-*Associative*: We must carefully distinguish true associations from accidental. Again use Figure 2.3 as an example. Phone number $e_{10}$ 1-800-555-1212 might occur very frequently with "Amazon Customer Service". However, this is just by random association since this phone number, being the general toll free number for US, also appears with other companies frequently. It is thus important to make "calibration" to purify the association we get.
- R-*Discriminative*: Intuitively, entity instances matched on more popular pages should receive higher scores than entity instances from less popular pages. This characteristic is especially useful when the document collection is of varying quality, such as the Web.

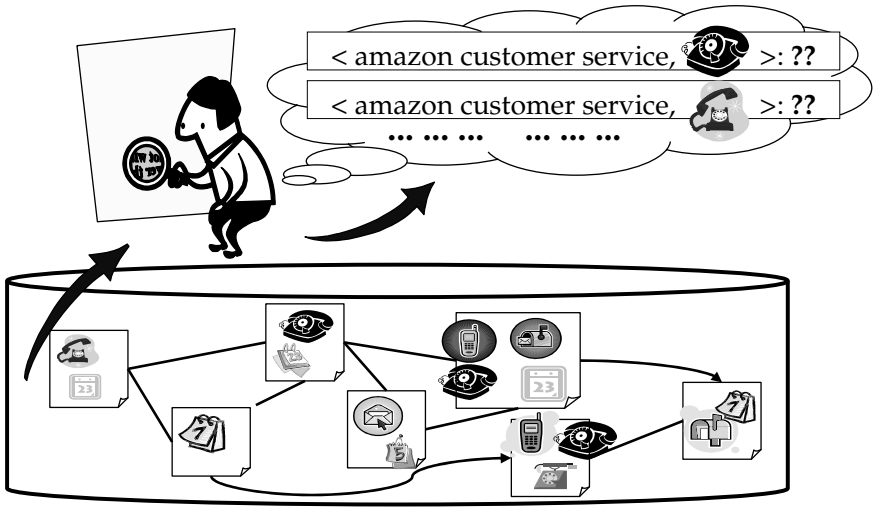



Figure 2.4: Impression Model: Conceptual Illustration

## 2.3 Conceptually: Impression Model

Toward a principled ranking model, we start with developing the insights– What is the *conceptual* model that captures the "ideal" behavior of entity search?

### 2.3.1 Impression Model

To begin with, assuming no resource or time constraints, we speculate, what is an *ideal* realization of entity search, over the Web as $\mathcal{D}$? To be concrete, consider query $Q1$ for finding tuple $\langle$"amazon customer service", #phone$\rangle$. As most users will probably do, we can access "amazon"-related pages (say, from a search engine), browse their contents, and follow links to read more, until we are satisfied (or give up), and returning our overall findings of promising #phone numbers.

Let's cast this process into an ideal execution, a conceptual model which we call the *Impression Model*, as Figure 2.4 shows. With unlimited time and resource, we dispatch an *observer* to repeatedly access the Web $\mathcal{D}$ and collect every evidence for substantiating any potential answer. This observer will visit as many documents and for as many times as he wishes. He will examine each such document $d$ for any #phone that matches $Q1$ (*i.e.*, following and near "amazon customer service") and form his judgement of how good the matches are. With an unlimited memory, he will remember all his judgements– *i.e.*, his *impression*. The observer will stop when he has sufficient impression, according to which he will score and rank each phone entity instance that occurs in $\mathcal{D}$.

To formalize this impression model, we take a probabilistic view to capture the observer's "impression." For a query $q$, given entity collection $E$ over document collection $\mathcal{D}$, Figure 2.5 sketches the impression

**The Observer's Impression Model**: *Conceptual Execution of Entity Search.*

**Given**: $\mathcal{E} = \{E_1, \ldots, E_N\}$ over $\mathcal{D} = \{d_1, \ldots, d_n\}$. Let $E = E_1 \times \ldots \times E_m$.
**Input**: $q = \alpha(E_1, \ldots, E_m, k_1, \ldots, k_l)$.

**0**: $\tau = 1$; /* *time tick.*
**1**: **while** ($\tau = \tau + 1$ **and** $\tau \leq T$):
**3**: $d^\tau =$ **access** document from $\mathcal{D}$; /* ***access layer.***
**4**: $\forall t \in E$: $Score(q(t)|d^\tau) =$ **recognize** $p(q(t)|d^\tau)$; /* ***recognizition layer.***
**5**: $\forall t \in E$: **output** $Score(q(t)) = \frac{\sum_{\tau=1}^{T} Score(q(t)|d^\tau)}{T}$; /* ***average impression.***

Figure 2.5: Impression Model: Basic Framework

framework. To execute entity search, at time $\tau$, the observer *accesses* a document, which we denote $d^\tau$, from $\mathcal{D}$– Let's abstract this mechanism as the *access layer* of the observer. Examining this $d^\tau$, he will *recognize* if any potential tuple $t$ occurs there. Let's abstract this assessment function as the observer's *recognition layer*. Formally, this assessment results in the *association probability* $p(q(t)|d^\tau)$– *i.e.*, how likely tuple $q(t)$ holds true given the "evidence" of $d^\tau$.

Eventually, at some time $\tau = T$, the observer may have sufficient "trials" of this repeated document visits, at which point his impression stabilizes (*i.e.*, with sufficient sampling from $\mathcal{D}$). To capture this convergence statistically, let's characterize the access layer by $p(d)$, the *access probability* of document $d$, *i.e.*, how likely $d$ may be drawn in each trial. Thus, over $T$ trials, $d$ will appear $T \cdot p(d)$ times. If $T$ is sufficiently large, the average impression (*i.e.*, statistical mean) will converge– which we similarly refer to as the *association probability* of $q(t)$ over $\mathcal{D}$:

$$p(q(t)|\mathcal{D}) = \lim_{T \to \infty} \frac{\sum_{\tau=1}^{T} p(q(t)|d^\tau)}{T} = \sum_{d \in \mathcal{D}} p(d) \cdot p(q(t)|d) \tag{2.1}$$

As this association probability characterizes how likely $t$ forms a tuple matching $q(t)$, when given the entire collection $\mathcal{D}$ as evidence, it is the "query score" we are seeking. While we will (in Section 2.4) further enhance it (with hypothesis testing), for now, we can view it as the final query score, *i.e.*,

$$Score(q(t)) = p(q(t)|\mathcal{D}).$$

The impression model provides a conceptual guideline for the entity search task, by a tireless observer to explore the collection for all potential entity tuples. While the framework is conceptually ideal, all the

key component layers remain open. We start with a "naive" materialization to motivate our full design.

### 2.3.2 Baseline: Naive Observer

As a first proposal, we develop the impression model with a simple but intuitive observer behavior, which uniformly treats every document in $\mathcal{D}$ and check if all entities and keywords are present. The final score is the aggregation of this simple behavior.

• *Access Layer*: The observer views every document equally, with a uniform probability $p(d) = \frac{1}{n}, \forall d \in \mathcal{D}$ (recall that $|\mathcal{D}| = n$).

• *Recognition Layer*: The observer assesses $p(q(t)|d)$ simply by the document "co-occurrence" of all the entity instances $e_i$ and keywords $k_j$ specified in $q(t)$: $p(q(t)|d) = 1$ if they all occur in $d$; otherwise 0.

• *Overall*: Filling the details into Eq. 2.1, we derive the score, or the expected impression, of a candidate tuple $t$ as follows:

$$Score(q(t)) = \sum_{d \in \mathcal{D}} \frac{1}{n} \cdot \begin{cases} 1 & \text{if } q(t) \in d \\ 0 & \text{otherwise} \end{cases} = \frac{1}{n} C(q(t)), \tag{2.2}$$

where $C(q(t))$ is the *document co-occurrence frequency* of $q(t)$, *i.e.*, the number of documents $d$ in $\mathcal{D}$ such that $q(t) \subseteq d$.

The naive impression model, while simple, intuitively captures the spirits of entity search– that of identifying entities from each documents *locally*, by the recognition layer, and aggregates across the entire collection *globally*, by the access layer. Overall, the naive approach results in using co-occurrence frequency of entities and terms as the query score of tuple $t$– a simple but reasonable first attempt.

As a starting point, to build upon the naive observer for a full realization of the ideal impression model, we ask: What are its limitations? As our checklist, we examine the five requirements as outlined in Section 2.2. The naive observer does meet the *holistic* requirement, as it aggregates the impressions across all documents– which is the essence of the impression model (and thus every materialization will satisfy). Systematically over the requirement list, we identify three limitations.

**Limitation 1:** *The access layer does not discriminate sources.* As R-*Discriminative* states, not all documents, as sources of information, are equal. However, with a uniformly-picking access layer, the naive observer bypasses R-*Discriminative*. It will thus not be able to leverage many document collections where

intrinsic (*e.g.*, link or citation structure) or extrinsic structures (*e.g.*, user rating and tagging) exist to discriminate documents as sources of information. How to enhance the access layer, so that documents are properly discriminated?

**Limitation 2:** *The recognition layer is not aware of entity uncertainty and conceptual patterns.* As R-*Uncertain* and R-*Contextual* mandate, entity instances are not perfectly extracted, and their matching with the query depends on keywords and other entities in the surrounding context. However, with an occurrence-only recognition layer, the naive observer does not respect either requirements. How to enhance the recognition layer, so that the tuple probabilities at each document are effectively assessed?

**Limitation 3:** *A validation layer is lacking.* As R-*Associative* states, our search should distinguish "intended" association of tuples from those "accidental" ones. The naive observer, however, believes in whatever "impression" he saw from the documents, and thus can be fragile regarding R-*Associative*. As we take a statistical view of the impression, we should equip the observer with statistical validation of his impression– or his "hypothesis"– and assess its significance. Our impression model, and therefore the naive observer, is missing such a critical *validation layer*.

## 2.4 Concretely: EntityRank

Upon the basic impression model (and the naive materialization), we now fully develop our entity-search scheme: *EntityRank*.

Motivated by the limitations of the basic model, we begin with presenting the complete *impression model* as Figure 2.6 shows. The full model completes the initial sketch in Figure 2.4 in two aspects: First, we add a "virtual observer" (at the right side), who will perform the same observation job, but now over a "virtual" collection $\mathcal{D}$' as a randomized version of $\mathcal{D}$. Second, we add a new validation layer to validate the impression of the real observer (over $\mathcal{D}$) by comparing it with that of the virtual observer (over $\mathcal{D}$'). Overall, our impression model consists of three layers:

- (Section 2.4.1) As Limitation 1 motivated, the *access layer* defines how the observer picks documents, and is thus responsible for *globally* aggregating tuple scores across the entire collection.
- (Section 2.4.2) As Limitation 2 motivated, the *recognition layer* defines how the observer examines a document, and thus considers *locally* assessing tuple probabilities in each document visited.
- (Section 2.4.3) As Limitation 3 motivated, the *validation layer* statistically validates the significance of the 'impression" by comparing it with the null hypothesis from the virtual observer.

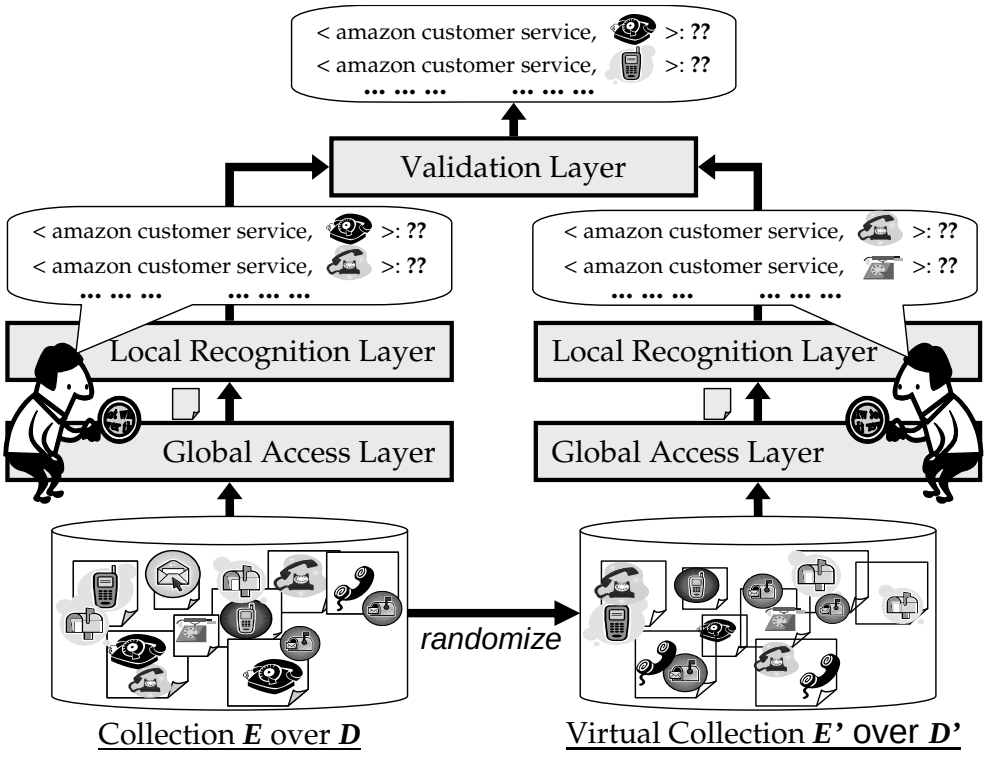



Figure 2.6: Impression Model: Complete Framework

This section will concretely materialize the three layers, to develop our *EntityRank* scheme. Figure 2.7 summarizes *EntityRank*. Overall, Equation (1) gives $Score(q(t))$, the scoring function of how tuple $t$ matches query $q$. Refer to our search task as Figure 1.3 defines, this scoring function determines the results of entity search. While we will develop step by step, as a road map, our objective is to, *first*, derive $p(q(t)|\mathcal{D})$ by materializing Eq. 2.1– which requires that we concretely define $p(d)$ and $p(q(t)|d)$. Sections 2.4.1 and 2.4.2 will develop these two parts respectively, which will together combine into $p(q(t)|\mathcal{D})$ given in Figure 2.7; we call this the *observed* probability, or $p_o$ for short. Section 2.4.3 will similarly derive the same probability for the random collection, which we call the *random* probability and denote $p_r$. *Second*, the final score $Score(q(t))$ is determined by comparing $p_o$ and $p_r$, in terms of hypothesis testing, which Section 2.4.3 will study.

While the *EntityRank* scheme satisfies all our "semantic" requirements of entity search– Is it admissible to efficient implementation? After all, we are pursuing entity search in the context of building a novel search engine, and efficiency is crucial for any online interactive search. Although query optimization techniques are beyond the focus of this chapter, we summarize our overall *EntityRank* algorithm and its implementation strategies in Section 2.4.4.

• **Query**: $q(\langle E_1, \ldots, E_m \rangle) = \alpha(E_1, \ldots, E_m, k_1, \ldots, k_l)$ **over** $\mathcal{D}$

• **Result**: $\forall\, t \in E_1 \times \cdots E_m$: Rank all $t$ by computing $Score(q(t))$ as follows.

(**1**) $Score(q(t)) = p_o \cdot \log \frac{p_o}{p_r}$, **where**

(**2**) $p_o \equiv p(q(t)|D) = \sum_{d \in D} \mathbf{PR}[d] \times \max_{\gamma} \left( \prod_{e_i \in \gamma} e_i.conf \times \alpha_B(\gamma) \times p(s|\gamma) \right)$

(**3**) $p_r \equiv p(q(t)|D') = \prod_{j=1}^{m} \left( \sum_{e_j \in d, d \in D} p(d) \right) \times \prod_{i=1}^{l} \left( \sum_{k_i \in d, d \in D} p(d) \right) \times \prod_{j=1}^{m} \overline{e_j.conf} \times \frac{\sum_s p(q(t)|s)}{|s|}$

Figure 2.7: *EntityRank*: The Scoring Function

### 2.4.1 Access Layer: Global Aggregation

The *access layer* defines how the observer selects documents, and is thus responsible for *globally* aggregating tuple scores across the entire collection. This layer must determine $p(d)$– how likely each document $d$ will be seen by the observer (and thus how $d$ will contribute to the global aggregation). As its objective (by R-*Discriminative*), this probability should discriminate documents by their "quality" as a source of information. The specific choice of an effective discriminative measure of quality depends on the document collection– what intrinsic structure or extrinsic meta data is available to characterize the desired notion of quality.

In the context of the Web, which is our focus, as a design choice, we decide to adopt the common notion of *popularity* as a metric for such quality. As the Web is a hyperlinked graph, the popularity of a page can be captured as how a page will be visited by users traversing the graph. Upon this view, we can materialize the access layer by the *random walk* model, where $p(d)$ can be interpreted as the probability of visiting a certain page $d$ in the whole random walk process. With its clear success in capturing page popularity, PageRank [12] is a reasonable choice for calculating $p(d)$ over the Web. That is, computing $\mathbf{PR}$ as the PageRank vector, as the overall result of the access layer, we have

$$p(d) = \mathbf{PR}[d]. \tag{2.3}$$

For this choice, we make two remarks: First, *like* in a typical search engine setting, this page discriminative metric can be computed *offline*. Second, *unlike* in a typical search engine, we are *not* using $\mathbf{PR}$ to directly rank result "objects" by their popularity. Instead, we are using these popularity values as a discriminative measure to distinguish each page as a *source* of information– For a tuple $t$, its score will aggregate, by the impression model (Eq. 2.1), over all the pages $d_i$ it occurs, each weighted by this $p(d_i)$ value.

We stress that, while we implement the access layer with "popularity" and in particular adopt PageRank, there are many other possibilities. Recall that our essentially objective is to achieve source discrimination– so that "good" pages are emphasized. First, such discrimination is not limited to link-based popularity– *e.g.*, user or editorial rating, tagging and bookmarking (say, del.icio.us), and query log analysis. Second, such discrimination may even be query-specific and controlled by users– say, to focus entity search on a subset of pages. For instance, we may restrict query $Q2$ and $Q3$ to only pages within the *.edu* domain; or, we may want to execute $Q4$ only for pages from amazon.com and bn.com. Such *corpus restriction*– with its access focus, not only speeds up search but also isolate potential noises that may interfere with the results. In general, our notion of the global access layer is to capture all these different senses of source discrimination.

### 2.4.2 Recognition Layer: Local Assessment

The recognition layer defines how the observer examines a document $d$, and thus accounts for locally assessing tuple probabilities in each document visited. Given a document $d$, we are to assess how a particular tuple $t$=$\langle e_1, \cdots, e_m\rangle$ matches the query $q(\langle E_1, \ldots, E_m\rangle) = \alpha(E_1, \ldots, E_m, k_1, \ldots, k_l)$. That is, this layer is to determine $p(q(t)|\ d)$, *i.e.*, how likely tuple $q(t)$, in the form of $\alpha(e_1, \ldots, e_m, k_1, \ldots, k_l)$, holds true, given $d$ as evidence. Consider $Q1$: Given $d$ as the snippet in Figure 2.3, for $t$=$\langle e_1$="800-201-7575"$\rangle$, the question becomes asking: $p(Q1(t)|d) = p(\mathsf{ow}(\text{amazon customer service } e_1) \mid d)$=?

To begin with, we note that a query tuple $q(t) = \alpha(e_1, \ldots, e_m, k_1, \ldots, k_l)$ may appear in $d$ in multiple occurrences, because each $e_i$ or $k_j$ can appear multiple times. For instance, in Figure 2.3, while $e_1$="800-201-7575" occurs only once, "amazon customer service" appears two times– so they combine into two occurrences. Let's denote such an occurrence of $(e_1, \ldots, e_m, k_1, \ldots, k_l)$ in document $d$ as $\gamma = (o_1, \ldots, o_n)$, where $n = m+l$; each $o_i$ is an *object occurrence*, *i.e.*, a specific occurrence of entity instance $e_i$ or keyword instance $k_j$ in $d$. *E.g.*, Figure 2.3 shows $(o_1, \ldots, o_4)$ as one occurrence for the above example.

For each potential tuple $t$, the recognition of the association probability $p(q(t)|d)$ must evaluate all its occurrences, and "aggregate" their probabilities– because each one contributes to supporting $t$ as a desired tuple. For this aggregation, we take the maximum across all occurrences of $t$ as its overall probability, *i.e.*,

$$p(q(t)|d) = \max_{\gamma \text{ is a tuple occurrence of } q(t)} p(\alpha(\gamma)) \tag{2.4}$$

With this aggregation in place, we now focus on the assessment of each occurrence $\gamma = \{o_1, \ldots, o_n\}$ for $p(\alpha(\gamma))$– *i.e.*, how this tuple occurrence matches the desired context pattern. This task involves two largely orthogonal considerations:

- Extraction uncertainty: For each $o_i$ that represents an entity instance $e_i$– Is it indeed an instance of type $E_i$? As Section 2.2 discussed, after performing extraction on $\mathcal{D}$ to prepare our "entity view" $E$, this probability $p(e_j \in E_j|d)$ is recoded in $e_i.conf$.
- Association context: Do the occurrences of $o_1$, ..., $o_n$ *together* match the pattern $\alpha$ that suggests their association as a desired tuple for query $q$?

With the two orthogonal factors, given $e_i.conf$, we can readily factor out the extraction uncertainty, and focus on the remaining issue: defining $p_{context}$– how $t$ matches $q$ in the context of $d$.

$$p(\alpha(\gamma)) \quad = \quad (\prod_{e_i \in \gamma} e_i.conf) \cdot p_{context}(\alpha(\gamma))$$

To determine $p_{context}$, this *contextual analysis* primarily consists of two parts, boolean pattern analysis and fuzzy proximity analysis, as we motivated in Section 2.2. Boolean pattern analysis serves the purpose of instantiating tuples, after which fuzzy proximity analysis serves the purpose of estimating in-document association strength of tuples.

**Context Operators $\alpha$.** We are to evaluate $p_{context}(\alpha(\gamma))$, to see how $\gamma$ occurs in a way matching $\alpha$, in terms of the surrounding context. Recall that, as Section 2.2 defined, in entity search, a query $q$ specifies a context operator $\alpha$, which suggests how the desired tuple instances may appear in Web pages.

As a remark, we contrast our usage of patterns in entity search with its counterparts in document search (e.g, current search engines). On the one hand, such pattern restriction is not unique in entity search. In typical document search, it is also commonly used– *e.g.*, a user can put "" around keywords to specify matching these keywords as a phrase. However, on the other hand, our entity search uses textual patterns in a rather different way– a tuple pattern $\alpha$ describes the possible appearance of the surrounding *context* of a desired tuple. In contrast, keyword patterns in document search are intended for the *content* within a desired document.

In this study, we treat a Web page as a linear sequence of words. As Section 2.2 mentioned, in the entity view, each entity or keyword occurrence $o_i$ is extracted with its positions at $o_i.pos$. An operator shall match on the *order* or *proximity* of objects: Each operator $\alpha$ applies to match an occurrence $\gamma$, *i.e.*, of the form $\alpha(o_1, \ldots, o_m)$. We will explain a few operators: doc, phrase, uw, ow.

We stress that, the context matching operator is for users (end-users or applications) to specify how the desired tuple may appear in documents. As in any query language, it also involves the *tradeoff* of simplicity (or ease of use) and expressiveness. We believe the exact balance of such tradeoff must depend on the actual application settings– Note that, while we develop it as a generic system, the entity search system can be

deployed in a wide range of settings, such as a general search engine for end users or a specialized vertical application, as Section 3.1 motivated.

Therefore, we advocate a two-fold strategy to balance the tradeoff, with a set of system *built-in* operators: On the one hand, for *simplicity*, while users may specify explicitly an $\alpha$ operator, the system must support a default when omitted (*i.e.*, order in our current system), as Section 3.1 shows (*e.g.*, ,$Q1$). On the other hand, for *expressiveness*, while an application may choose to use our supported operators, the system must support a *plug-in* framework for applications to define their own specific context patterns.

In what follows, we explain some supported operators, each in the form of $\alpha(o_1, \ldots, o_m)$. Our purpose is to demonstrate how an operator is defined, in order to be plugged into the recognition layer. Each operator consists of two parts (as R-*Contextual* states), *i.e.*, the *pattern* constraint $\alpha_B$ and *proximity* function $\alpha_P$. Thus, we can further define our context probability (of Eq. 2.5) as

$$p_{context}(\alpha(\gamma)) \equiv \alpha_B(o_1, \ldots, o_m) \cdot \alpha_P(o_1, \ldots, o_m).$$

**1. Boolean Pattern Qualification.** As the first step, $\alpha$ will qualify, by a constraint $\alpha_B$ that returns 1 or 0 (*true* or *false*), whether some pattern constraint on the order and adjacency of $o_i$ is satisfied.

- doc($o_1$, ..., $o_m$): objects $o_i$ must all occur in the same document.
- phrase($o_1$, ..., $o_m$): objects $o_i$ must all occur in exactly the same sequence (*i.e.*, order and adjacency) as specified.
- uw($n$)($o_1$, ..., $o_m$): objects $o_i$ must all occur in a window of no more than $n$ words; $n$ default as the document length.
- ow($n$)($o_1$, ..., $o_m$): in addition to uw, $o_i$ must all occur in the order.

**2. Probabilistic Proximity Quantification.** As the second step, the operator $\alpha$ will quantify, by a probability function $\alpha_P$, how well the proximity between objects match the desired tuple– *i.e.*, how the objects' positions indicate their association as a tuple, once they are qualified (above). Essentially, each operator will thus define a probabilistic distribution that assesses such association probabilities given object positions.

We propose an intuitive and practical model, the *span proximity model*, to capture our basic intuition that the closer they appear to each other, the more likely they are associated with each other. While our system currently only supports the span proximity model, our experimental results show that it is effective for a wide range of scenarios.

In the span model, we characterize the proximity strength of a tuple occurrence $\gamma = (o_1, \ldots, o_m)$ as depending on its *span* length– the shortest window that covers the entire occurrence, denoted by $s$. We define the context association probability of $\gamma$, *i.e.*, how $e_i$ and $k_j$ associate into a tuple, as solely determined by span.

$$\alpha_P(\gamma) \equiv p(\gamma \text{ is a tuple}|s), \text{or simply } p(\gamma\ |s)$$

Finally, we must estimate the distribution $p(\gamma|s)$. We can "learn" this distribution by collecting some true tuples (as our "labelled data")– *i.e.*, $\gamma$'s that are really tuples. With sufficient examples, we can obtain an empirical distribution of, for real tuples, how their spans vary, *i.e.*, $p(s|\gamma)$. In our current implementation, we obtain our span distribution as Figure 2.8 shows, using labelled tuples that are company names and their phone numbers (*e.g.*, "IBM", "877-426-2223") from the Web corpus. Finally, by Bayes' theorem,

$$p(\gamma\ |s) = \frac{p(\gamma)}{p(s)} p(s|\gamma) \propto p(s|\gamma),$$

where we remove the prior probabilities $p(\gamma)$ and $p(s)$: Note that, for practical implementation, as the priors are hard to measure, we assume they are the same, for all different $\gamma$ and $s$. Thus, these constant priors will not affect ranking of entities.

Putting together the derivations so far back to where we started, *i.e.*, Eq. 2.4, as the overall result of the recognition layer, we obtain:

$$p(q(t)|d) = \max_{\gamma} \prod_{e_i \in \gamma} e_i.conf \cdot \alpha_B(\gamma) \cdot p(s|\gamma) \tag{2.5}$$

As a related approach, Chakrabarti et al. [19] propose a *discriminative* model to measure the association between an entity instance and keywords within a document. Our span proximity model is probabilistic– it bridges the proximity between objects to their probability of associating into a tuple. By relating each span with a probability, we can thus integrate our local recognition model with the global access model in a probabilistic framework seamlessly.

Ideally, it would be best to learn such a span model for each kind of query– because different types tuples may vary in their spans. However, it is impractical to do so due to the unpredictable and large number of different queries. It is possible to classify queries into different categories and learn a approximate span model for each category. In this chapter, we approximate this by using the one span model we learned in Figure 2.8. Although our "span" model may not be as accurate as other more sophisticated local models, such as the one presented in [19], we believe it still captures the basic insights of a local recognition model.

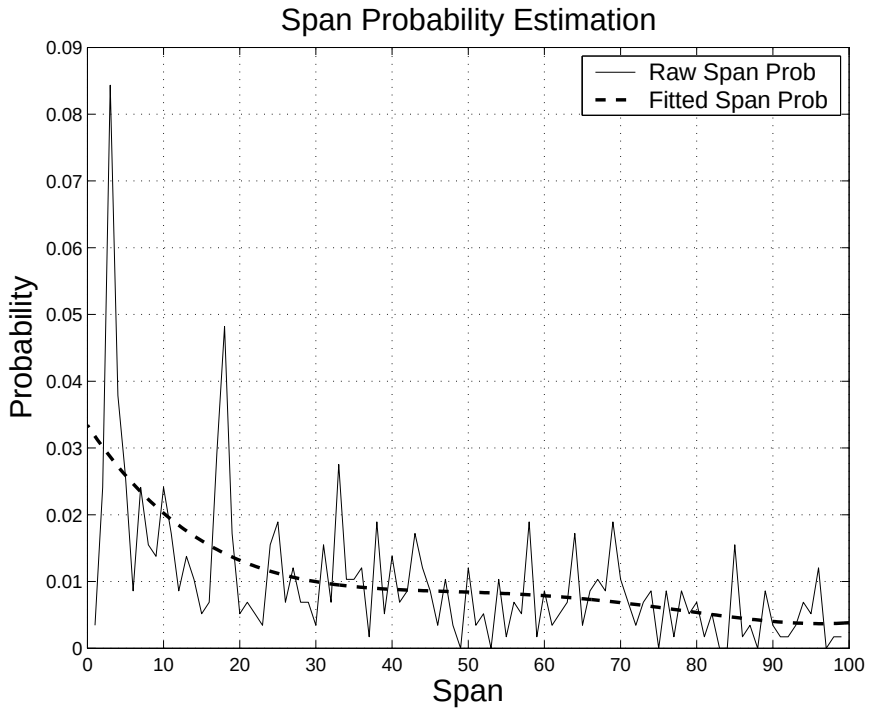



Figure 2.8: The Span Proximity Model

We stress that our recognition layer is a general framework that can plug-in any pattern and proximity models. Our current choice of basic patterns and the span model is just one reasonable possibility. They are robust and easy to apply, which gives us fast online processing.

### 2.4.3 Validation Layer: Hypothesis Testing

As a new layer, the *validation layer* statistically validates the significance of the "impression." Our approach is by comparing it with the null hypothesis– simulating an observer over a virtual collection. We note that the construction of a random collection $\mathcal{D}$' is only conceptual– it will derive a formula that can be efficiently computed (Section 2.4.4) without materializing $\mathcal{D}$'.

Since our impression model dispatches the observer (Figure 2.6) to collect his impression of entity associations, we must ask– Are these associations significant? Recall from Eq. 2.1 that $p(q(t)|\mathcal{D})$ is an average impression, *i.e.*, the statistical mean, over $\mathcal{D}$. By taking a probabilistic view, we are now equipped with principled statistical tools to "validate" if the impression of association, as our *hypothesis* to test, is significant. As the *null hypothesis*, we suppose the associations are "unintended"– *i.e.*, as the result of randomly associating entities and keywords in a randomized collection $\mathcal{D}$'. By contrasting observed probability $p_o$ with random probability $p_r$, we will obtain a final score $Score(q(t))$ that captures the significance of our hypothesis– *i.e.*, intended tuple association.

**Null Hypothesis: Randomized Associations.** As our null hypothesis, we suppose that the association of $t = (e_1, \ldots, e_m, k_1, \ldots, k_l)$ is simply accidental– instead of semantically intended as a tuple. To simulate the outcome of the null hypothesis, we must create $\mathcal{D}$' as a randomized version of $\mathcal{D}$. In comparison, $\mathcal{D}$' should resemble $\mathcal{D}$ as much as possible, except that in $\mathcal{D}$' the associations of keywords and entities are purely random. The creation of a random document $d'$ in $\mathcal{D}$' is as follows:

First, we will randomly draw entities and keywords into $d'$. Each $e_i$ or $k_j$ will be drawn independently, with a probability the same as that of appearing in any document of $D$. We thus preserve the individual entity/keyword probabilities from $D$, but randomize their associations. This probability can be derived as below, which ensures that the probability of observing a keyword/entity instance in the process of visiting the $\mathcal{D}$' through a uniform access layer will be the same as observing it in the original $D$.

$$p(e_i \in d') = \sum_{e_i \in d, d \in D} p(d); \quad p(k_j \in d') = \sum_{k_j \in d, d \in D} p(d)$$

Further, if a keyword/entity instance is drawn to appear in virtual document $d'$, its position in the document will also be randomly chosen. We also preserve the entity extraction probability of entity instances, by using the average extraction probability $\overline{e_j.conf}$ (the average over all the extraction probabilities of all the occurrences of entity instance $e_j$ in $D$) of entity instance $e_j$ as the extraction probability of all the occurrences of entity instance $e_j$ in $D'$.

Supposing we construct $\mathcal{D}$' with all such random documents $d'$, we ask: What will be the "average impression" that our observer will perceive in this random collection? By Eq. 2.1, this average impression is the summation over all documents. We can simplify it by noting that 1) if $q(t)$ does not even appear in $d'$, then $p(q(t)|d') = 0$, 2) otherwise, $p(q(t)|d')$ has the same value– since we create all $d'$ in exactly the same way. Consequently, we get:

$$\begin{aligned} p(q(t)|D') &= \sum_{d' \in \mathcal{D}' \textbf{and } q(t) \in d'} p(d') \cdot p(q(t)|d') \\ &= p(q(t)|d') \cdot \sum_{d' \in \mathcal{D}' \textbf{and } q(t) \in d'} p(d') \\ &= p(q(t)|d') \cdot p(q(t) \in d') \end{aligned} \tag{2.6}$$

Here, $p(q(t) \in d')$ is the probability of $t$ appearing in some $d'$. As keywords and entity instances are drawn independently into $d'$, this probability is simply the product of the probabilities of each keyword and entity instances appearing in $d'$, *i.e.*,

$$p(q(t) \in d') = \prod_{j=1}^{m} p(e_j \in d') \prod_{i=1}^{l} p(k_i \in d').$$

Next, we derive the random association probability, $p(q(t)|d')$, of tuple $t$ in document $d'$. As we discussed in Section 2.4.2, it is the product of entity probability and contextual probability:

$$p(q(t)|d') = (\prod_{j=1}^{m} \overline{e_j.conf}) \cdot p_{context}(q(t)|d').$$

The contextual probability $p_{context}(q(t)|d')$ is dependent on the span of tuple $t$, as Section 2.4.2 discussed. As keywords and entity instances appear randomly in $d'$, the possible spans of tuple $t$ should also appear randomly. This means different span values are equally likely. Thus, we can use the average of all the span probabilities (*e.g.*, as Figure 2.8 shows) for this probability estimation:

$$p_{context}(q(t)|d') = \overline{p}(q(t)|s) = \frac{\sum_s p(q(t)|s)}{|s|},$$

where $|s|$ refers to the number of distinct span values.

Putting together these derivations back to Eq. 2.6, we will obtain $p(q(t)|\mathcal{D}')$, which Figure 2.7 shows as the random probability $p_r$.

**Hypothesis Testing: G-Test.** To judge if the association of $t$ is statistically significant, we should compare the *observed* $p_o$ versus the *random* $p_r$, which we have both derived in Figure 2.7. We can use standard statistics testing, such as mutual information, $\chi^2$, G-test [33], *etc.*. Our implementation adopts G-test to check whether $p_o$ indicates random association or not,

$$Score(q(t)) = 2(p_o log\frac{p_o}{p_r} + (1-p_o) log\frac{1-p_o}{1-p_r}) \tag{2.7}$$

To interpret, we note that the higher the G-test score is, the more likely entity instances $t$ together with keywords $k$ are truly associated. We take this score for the final ranking of the tuples. In practice, given a large web page collection containing many keywords and entity instances, the chance that an entity instance occurs in the collection is extremely small. Hence, $p_o, p_r \ll 1$, Eq. (2.7) can be simplified as:

$$Score(q(t)) \propto p_o \cdot log\frac{p_o}{p_r} \tag{2.8}$$

### 2.4.4 EntityRank: Implementation Sketch

We have developed *EntityRank*, as a concrete materialization of the impression model, and it satisfies all our requirements (Section 2.2) of entity search– but, can it be efficiently implemented for online interactive search– at a speed comparable to current page-oriented search engines? Our analysis of the scheme

(Figure 2.7) shows that, in fact, the *EntityRank* framework can be easily built upon current engines, thus immediately leveraging the impressive infrastructure and scalable efficiency already in place. While this chapter focuses on the ranking scheme, and not query processing, we show how *EntityRank* can be efficiently realized.

**Why Feasible?** Let's examine how we may realize *EntityRank*. In Figure 2.7, $Score(q(t))$ has two components, $p_o$ and $p_r$:

First, $p_r$ can mostly be pre-computed off-line (before query), and thus its cost is negligible. There are four factors: Factors 1 and 2 are the "document frequencies" of entities $e_j$ and keywords $k_i$. We can pre-compute them for every term; when the query arrives, we simply lookup in a table. We can similarly handle factors 3 and 4.

Second, $p_o$ boils down to "pattern matching," which is a major function of today's page-based search engine. The first factor requires PageRank, which can be done off-line. The second factor requires matching specific tuple occurrences $\gamma$ (Section 2.4.2), which can only be executed when the query terms (*e.g.*, "amazon" and #phone) and patterns (*e.g.*, ow) are specified. Nevertheless, such pattern matching is well supported in current engines, by using inverted lists– our realization can build upon similar techniques.

**Possible Implementation.** To begin with, we assume that a document collection $\mathcal{D}$ has been transformed by way of entity extraction into an entity collection $E$ with entities $\{E_1, E_2, \ldots, E_n\}$.

• *Indexing:* To support entity as a first-class concept in search, we index entities in the same way as indexing keywords.

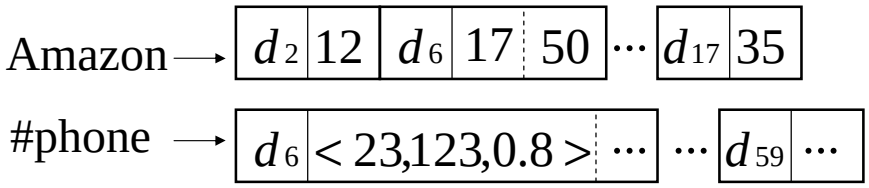



Figure 2.9: A Snippet of Index

We use the standard inverted index for indexing keywords. We index the extracted entities similar to keywords. To index an entity type $E_i$, our system will produce a list containing all the information regarding the extracted instances of $E_i$. Specifically, the list records for each occurrence of entity instance $e_i$, the position $e_i.pos$ of the extraction in the documents, *e.g.* position 23 at document $d_6$, the entity instance ID $e_i.id$, *e.g.* ID 123 for representing phone number "805-213-4545", and the confidence of extraction $e_i.conf$, *e.g.* 0.8. All the occurrence information of entity instances of a certain entity type is stored in an ordered list according to document number as shown in Figure 2.9.

**The EntityRank Algorithm**: *Actual Execution of Entity Search.*

**Given**: $L(E_i), L(k_j)$: Ordered lists for all the entity and keywords.
**Input**: $q = \alpha(E_1, \ldots, E_m, k_1, \ldots, k_l)$.

**0**: Load inverted lists: $L(E_1), \ldots, L(E_m), L(k_1), \ldots, L(k_l)$ ;
/* *intersecting lists by document number*
**1**: **For** each doc $d$ in the intersection of all lists
**2**: Use pattern $\alpha$ to instantiate tuples; /* *matching*
**3**: **For** each instantiated tuple $t$ in document $d$
**4**: **Calculate** $p(q(t)|d)$ ; /* *Section 4.2*
**5**: **For** each instantiated tuple $t$ in the whole process
**6**: **calculate** $p(q(t)|D) = \sum_d p(q(t)|d)p(d)$; /* *observed probability*
**7**: **output** $Score(\mathrm{q(t)}) = p(q(t)|D) \log \frac{p(q(t)|D)}{p(q(t)|D')}$; /* *Section 4.3*

Figure 2.10: The *EntityRank* Execution Framework

• *Search:* Figure 2.10 shows the pseudo code of our *EntityRank* algorithm for supporting entity search.

Let's walk through the algorithm for query "ow(amazon #phone)" upon the index in Figure 2.9. We first load the inverted list for keyword "Amazon" and entity type "#phone" (line 0). Then we iterate through the two lists in parallel , checking the intersection of documents (line 1). In this example, the algorithm will first report $d_6$ as the intersecting document. Then the algorithm will further check the postings of keywords and entity types to see if the specified query pattern "ow" is satisfied in $d_6$ (line 2). In this case, entity instance with ID "123" at position 23 is matched with keyword "Amazon" at position 17 (not position 50). A tuple of entity instance with ID "123" is therefore instantiated, and its local association probability will be calculated according to the local recognition layer 2.4.2 (line 3 and 4). All the instantiated tuples and their local scores are stored for later on aggregation and validation purposes. Once the parallel scan of lists ends, we can sum up all the local scores (multiplied by the popularity of document) into forming the observed average association probability for each tuple $t$ (line 6). As just discussed, the cost of $p_r$ is negligible.

Observe that the core of our *EntityRank* algorithm (line 1-4) is essentially sort-merge-join of ordered lists. This only requires scanning through all the lists once, in parallel. Therefore, the algorithm is linear in nature and could be run very efficiently. Moreover, this sort-merge-join operates on a document basis. This implies that this procedure (line 1-4) can be fully parallelized, by partitioning the collection into sub-collections. This parallelism allows the possibility of realizing entity search in very large-scale, supported by a cluster of nodes. Our prototype system, to be discussed in Section 2.5.3, leverages such parallelism on a cluster of 34

nodes.

## 2.5 Prototype and Experiments

This section first discusses the prototype system we built for supporting entity search. Then we describe a few applications that could be easily built upon on system, which clearly show the usefulness of entity search qualitatively. Finally, we use our large-scale experiment to quantitatively verify the effectiveness of our ranking algorithm *EntityRank*, as compared to other ranking schemes.

### 2.5.1 System Prototype

We build our entity search system upon the Lemur IR Toolkit. We mainly morphed the indexing module of the Lemur Toolkit to be able to index entities in addition to keywords and implemented our own query modules for supporting *EntityRank*.

For getting data from the Web, we obtained our corpus from the Stanford WebBase Project, as our "virtual" Web Crawler.

For entity extraction, we have implemented two types of entity extractors. First, our *rule-driven* extractor tags entities with regular patterns– *e.g.*, #phone entity and #email entity, etc. Second, our *dictionary-driven* extractor works for entities whose domains, or *dictionaries* are enumerated– *e.g.*, #university entity, #professor entity, #research entity(as areas in CS), *etc.*.

We discuss the architecture of our prototype entity search system, whose components could be divided into two categories: offline processing and online processing (*i.e.* entity ranking). The core components are shown in Figure 2.11.

#### Offline Processing

Entity extraction and indexing are the two main offline processing modules for entity search, as shown in the dashed box in Figure 2.11.

**Entity Extraction:** Entity extraction, which has been extensively studied, aims to extract entity instances from documents. It can be either straightforward, *e.g.* using regular expression for email address extraction, or very sophisticated, *e.g.* using statistical classifiers for street address identification. As entity extraction is inherently imperfect, which is unavoidable for complicated extraction tasks, our entity search must essentially deal with uncertainty. By this offline extraction, we will recognize, for each entity instance $e_i$, all

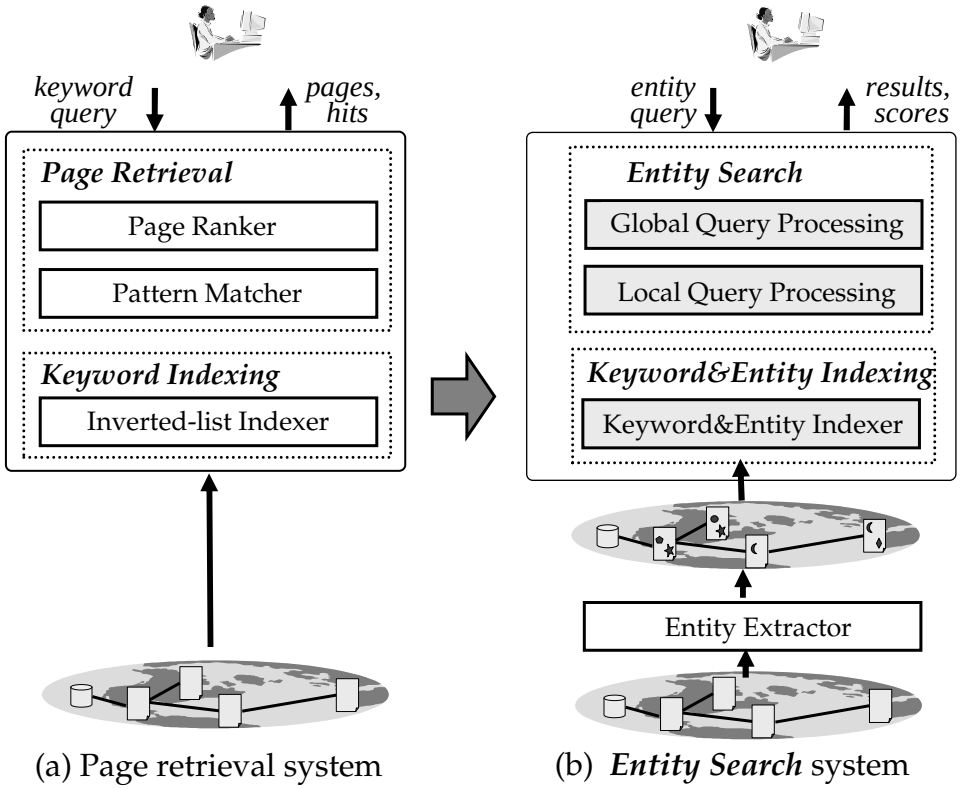



Figure 2.11: System Architecture

its *occurrences* in the corpus. To facilitate query matching, we will record the "features" of each occurrence $e_i$– These local occurrence features will facilitate our local scoring to quantify the strength of local matchings.

- Position $e_i.pos$: the document id and word offset of this instance occurrence, *e.g. instance* $e_1$ may occur at ($d_6$, 23).
- Entity ID $e_i.id$: the unique id that represents the string value of an entity instance, *e.g. instance* $e_1$ with ID 123.
- Confidence $e_i.conf$: the estimated probability that shows how likely this entity instance occurrence is regarded as an instance of entity type $E_i$, *e.g. instance* $e_1$ with extraction confidence 0.80.

An important aspect in entity extraction is entity disambiguation, for instance, figuring out that two phone numbers "213-4545" and "805-213-4545" actually refer to the same phone number. Entity disambiguation is beyond the scope of this study. We believe this is an orthogonal issue to the task of entity search. However, relying on large scale document analysis where much redundancy exists, we can afford not performing sophisticated entity disambiguation. For example, the phone number of "805-213-4545" appears frequently on the Web and could provide enough evidence for ranking.

**Indexing:** To support entity as a first-class concept in search, we index entities in the same way as indexing keywords.

To index the extracted entity instances, the indexer builds an inverted index of entities, in addition to the traditional keyword inverted index. Given an entity type $E_i$, the *entity inverted index* will return a list containing all the information regarding the extracted instances of $E_i$. Specifically, the *entity inverted index* records for each occurrence of entity instance $e_i$, the position $e_i.pos$ of the extraction in the documents, the entity instance ID $e_i.id$, and the probability of extraction accuracy $e_i.conf$, *e.g.* an occurrence of phone instance "805-213-4545" as ($d_6$, 23, 123, 0.8). Such information is stored in an ordered list according to the document ID, similar to a keyword inverted index. All the occurrences of keywords and entity types are recorded in their respective inverted indices. Therefore, given a query consisting of keywords and entity types, our system only need to load and process the inverted lists for each keyword and each entity type, respectively.

In addition to entity extraction and indexing, there are other tasks that need to done offline before online queries are executed. For example, the popularity score of each document should be computed offline. The individual frequency of keywords and entity instances in a corpus should also be computed in an offline fashion.

### Online Processing

In this subsection, we discuss the two main online processing modules corresponding to the local scoring and global scoring of our ranking model, as shown in the solid box in Figure 2.11.

**Local Query Processing:**

First, we use a pattern matcher to instantiate tuples by matching pattern $\alpha(K_1, \ldots, K_l, E_1, \ldots, E_m)$. The matcher will retrieve the inverted lists of all the entity types $E_i$ and keywords $K_j$, and then perform one pass of "sort-merge join" to match tuples. That is, the matcher will iterate through all the lists in parallel. For every document $d$ in the intersection, the matcher will check the actual positions of every $E_i$ instance $e_i$ and keywords $k_j$, to determine whether pattern $\alpha$ is satisfied. If it is, a tuple is instantiated and its local score is calculated according to the extraction confidence of its entity instances and the contextual relationship between its entity instances and keywords (more concretely their positions in the document). Then the matched tuple, with its local score, is sent to the global query processing module for aggregation.

In terms of computation, in spirit, it is the same as what needs to be done in traditional document retrieval. The operation, "sort-merge join", used in answering almost all the document retrieval queries, is well known to be easy to compute in a very fast manner due to its linear computational nature.

This local query processing module operates on a document basis, and therefore could be fully parallelized by partitioning the whole document collection into sub-collections. This query processing feature enables supporting entity search in large-scale, by utilizing the power of disturbed computing.

**Global Query Processing:**

The global query processing module, upon getting a user's entity search query, sends the query to distributed local query processing nodes. It then waits for the local processing nodes to produce matched tuples with their local scores. After receiving all the tuples with their local scores, it will perform global aggregation, by considering factors such as the popularity of pages, the frequency of individual keywords and entity instances, etc in the aggregation process. The goal is to derive a final score for each distinctive tuple that truly captures the association between entity instances and keywords in the tuple. Finally the tuples are ordered by their global score and output to the user. For concise presentation purposes, only entity instances of each tuple need to be shown, as the keywords for every tuple are the same.

Standard network protocol could be used for the communication between the global processing unit and the local processing units. It is worth noticing that almost all the expensive computations, the IO intensive ones that access indices, are done locally. The global processing workload is light, given all the information is readily available and simple aggregation could be performed very quickly.

For more details regarding the prototype system, please refer to our work [25] on the overall system architecture for entity search.

### 2.5.2 Qualitative Analysis: Case Studies

This subsection studies some of the possible applications that could be built upon entity search to show its promise qualitatively.

**Question Answering: Scenario 1**

Entity search could be a good candidate as the core search component for supporting question answering tasks. By using existing techniques, such as removing stop words, identifying the entity type for the question, we could effectively turn a lot of questions into entity search queries supported by our system.

Towards this goal, we built a yellowpage application with the following setting:

- Entity collection: $E = \{\#\mathsf{phone}, \#\mathsf{email}\}$
- Document collection: $\mathcal{D}$ = the Web

Such a system addresses our motivating query $Q1$, finding the Costumer Service number of Amazon, in Section 3.1.

| | | |
|---|---|---|
| Chris Clifton | Purdue Univ | Data Mining |
| Sunil Prabhakar | Purdue Univ | Database Systems |
| Jiawei Han | UIUC | Data Mining |
| David J. Dewitt | Univ of Wisc | Database Systems |

Table 2.1: Professors in DB-related Areas (Partial)

| | |
|---|---|
| sigmod04-040611.pdf | sigmod04-040617.ppt |
| URL: http://www.cs.ucsb.edu/~su/tutorials/sigmod2004.html | |
| publications/tr-db-01-02.pdf | publications/sigmod2001.ppt |
| URL: http://www.ics.uci.edu/~iosif/index.html | |

Table 2.2: Pairs of PDF and PPT Files for SIGMOD (Partial)

**Relation Discovery: Scenario 2&3**

Our query primitive and ranking algorithm for entity search could afford more than one entity at a time. Such queries, containing multiple entities, could be viewed as relational discovery queries.

Towards this goal, we built an application on Computer Science domain with the following setting:

- $E = \{\#\mathsf{professor}, \#\mathsf{research}, \#\mathsf{university}, \#\mathsf{pdf\_file}, \#\mathsf{ppt\_file}, \#\mathsf{phone}, \#\mathsf{email}\}$
- $\mathcal{D}$ = a collection of computer science related webpages

This application could answer user queries $Q2$ and $Q3$ we raised in scenario 2 and 3 in Section 3.1. Partial results are shown in Table 2.1 for query $Q2$ and in Table 2.2 for query $Q3$ respectively.

**Information Integration: Scenario 4**

Entity search could also be a good candidate for supporting ad-hoc on-the-fly information integration, whose goal is to assemble different attributes (entities) together into one relation.

Towards this goal we built an online book shopping application based on the query results returned from multiple deep web sources in the Book Domain.

- $E = \{\#\mathsf{title}, \#\mathsf{author}\, \#\mathsf{date}, \#\mathsf{price}, \#\mathsf{image}\}$
- $\mathcal{D}$ = result pages returned from deep Web sources regarding book queries

Users can ask queries regarding possible combinations of keywords and the entity types. Figure 2.12 shows the result for a query that tries to find images of books with keyword "Hamlet" in title, motivated by query $Q4$ we discussed in Section 3.1.

For all the four scenarios, we have built applications with different datasets and witnessed great usefulness of entity search in all of them. In particular, we will systematically evaluate Scenario 1 because it has a large realistic corpus. For all other scenarios, the results were also clearly promising: For instance, for queries

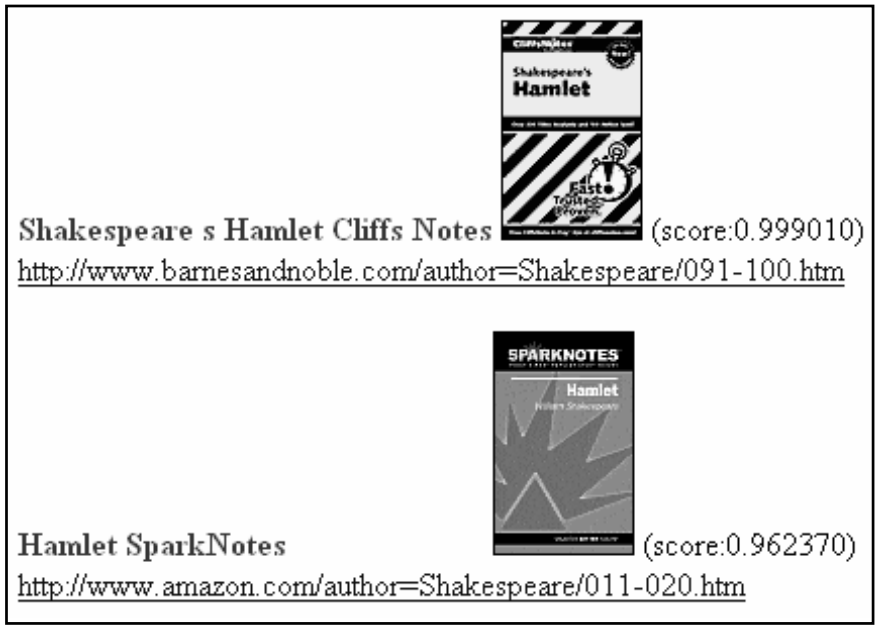



Figure 2.12: Images of Books with "Hamlet" in Title (Partial)

finding relations $\langle professor, research \rangle$ and $\langle professor, email \rangle$, when counting the top-match research area and email for each professor, the result achieves between 80% - 90% precision and recall.

### 2.5.3 Quantitative Systematic Evaluation

In order to demonstrate the effectiveness of our ranking algorithm *EntityRank* for supporting entity search, we have build a large scale, distributed system on a real Web corpus. In this subsection, we will first briefly discuss the setup of our system. Then, we will use typical query sets to show the accuracy of our ranking model over other ranking schemes.

#### Experiment Setup

To empirically verify that our ranking model is effective for supporting entity search, we decide to use the Web, the ultimate information source, as our corpus. Our corpus, a general Web crawl in Aug, 2006, is obtained from the WebBase Project. The total size is around 2TB, containing 48974 websites and 93 million pages.

To process such terabyte-sized data set, we ran our indexing and query processing modules on a cluster of 34 machines, each with Celeron 2.80GHz CPU, 1 GB memory and 160 GB of disk. We evenly distribute the whole corpus across these nodes.

On this corpus, we target at two entity types: phone and email. They are extracted based on a set of

regular expression rules. We extracted around 8.8 million distinctive phone entity instances and around 4.6 million distinctive email entity instances.

**Accuracy Comparison**

To get a first feeling of our ranking model, we tested a few finding-phone-number and finding-email-address queries, that are similar to our motivating scenario 1 in Section 3.1.

In addition to analyzing the results returned by *EntityRank*, we also try to compare our results with the following five approaches:

- N (Naive Approach): Tuples are ranked according to the percentage of documents in which they are matched.
- L (Local Model Only Approach): Tuples are solely ranked using their highest observed local association probability.
- G (Global Aggregation Only Approach): Tuples are ranked according to the summation of the score of documents in which they are matched. Pagerank score is used as the document score.
- C (Combination of Local Model and Global Aggregation Approach): Tuples are ranked according to the summation of the local association probabilities from matched documents.
- W (*EntityRank* Without G-test): Tuples are ranked according to their observed association probability, without performing the G-test for validation. The purpose of testing this approach is to see the effect of hypothesis test in the ranking framework.

| Query | Telephone | ER | L | N | G | C | W |
|---|---|---|---|---|---|---|---|
| Citibank Customer Service | 800-967-2400 | **1** | 4 | 7 | 43 | **1** | **1** |
| New York DMV | 800-342-5368 | **2** | **2** | 213 | 882 | 5 | 3 |
| Amazon Customer Service | 800-201-7575 | **1** | **1** | 52 | 83 | **1** | **1** |
| Ebay Customer Service | 888-749-3229 | **1** | 7 | 859 | 118 | 2 | 13 |
| Thinkpad Customer Service | 877-338-4465 | 5 | 12 | 249 | 127 | 19 | **4** |
| Illinois IRS | 800-829-3676 | **1** | **1** | 157 | 697 | 3 | 2 |
| Barnes & Noble Customer Service | 800-422-7717 | **1** | 2 | 2158 | 1141 | 7 | **1** |

Figure 2.13: Telephone Number Queries

The results of executing motivating queries using different ranking models are shown in Figure 2.13 and 2.14. The first column lists keywords used in the query. The second column lists the correct entity instance returned for each query (manually verified). The third, fourth, fifth, sixth and seventh columns list the rank of the correct phone number in results by the various approaches described above respectively.

| Query | Email | ER | L | N | G | C | W |
|---|---|---|---|---|---|---|---|
| Bill Gates | bgates@microsoft.com | **4** | 44 | 2502 | 376 | 21 | 23 |
| Oprah Winfrey | oprah@aol.com | **2** | 6 | 745 | 80 | 4 | 3 |
| Elvis Presley | elvis@icomm.com | **5** | 56 | 1106 | 267 | 20 | 8 |
| Larry Page | larrypage@google.com | **8** | 24 | 9968 | 26932 | 12 | 11 |
| Arnold Schwarzenegger | governor@governor.ca.gov | **4** | 45 | 165 | 169 | 5 | 6 |

Figure 2.14: Email Queries

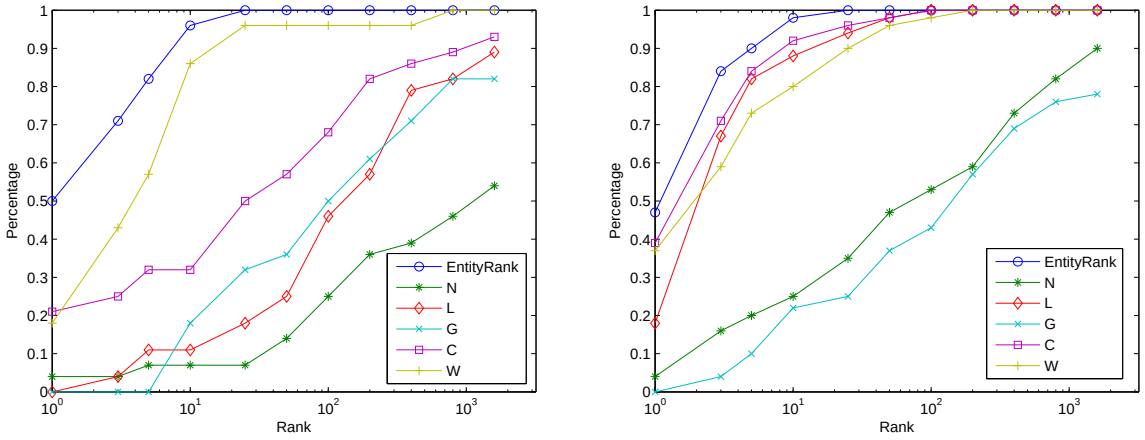
As we can see, *EntityRank* (ER) consistently outperforms other ranking methods. Almost all the right answers are returned within top 3 for finding phone numbers and more than half of the right answers are returned within top 4 for finding email addresses.

To study the performance of our method in a more systematical way, we use the following ways to come up with typical queries for each query types. **Query Type I (phone)**: We use the names of top 30 companies in Fortune 500, 2006 as part of our query, together with phone entity type in the query. **Query Type II (email):** We use the names of 88 PC members of SIGMOD 2007 as part of our query, together with email entity type in the query. 37 out of the 88 names that don't have any hit with any email entity instance are excluded. This is due to the reason that our corpus is not complete (2TB is only a small proportion of the whole Web).

To measure the performance, we use the mean reciprocal rank ($MRR$) as our measure. This measure essentially calculates the mean of the inverse ranks of the first relevant answer according to queries. The value of this measure lies between 0 and 1, and the higher the better. As it is time consuming to manually check all the returned results to identify the first relevant tuple in each result, especially given the fact that in lots of the results the first related entity tuple appears very high in rank. We come up with the following two approximate methods for estimating the rank where the first related entity tuple is returned.

In evaluation method 1, we first manually go through the result returned by our *EntityRank* algorithm, finding the first related tuple (usually within top 5). Then, we look up the place where this tuple appears using other ranking algorithms. Although this method is biased towards our *EntityRank* algorithm, it still makes sense as intuitively the related tuple (manually verified) returned by *EntityRank* should also be ranked high using other methods. At minimum, this evaluation method gives a meaningful lower bound of the $MRR$, referred as $\lfloor MRR \rfloor$, of the first relevant tuple.

Evaluation method 2 is a much more conservative evaluation method. We manually check the top 10 entries of the results returned by each ranking algorithm. If a relevant tuple is found within top 10 entries, we will record the rank, otherwise, we will just use rank 10. The intuition of using this method is that




(a) Query Type I

(b) Query Type II

Figure 2.15: Satisfied Query Percentage under Various Ranks

normally the top 10 results (in the first pages) are most important for users. This evaluation method gives a meaningful higher bound of the $MRR$, referred as $\lceil MRR \rceil$, of the first relevant tuple.

Figure 2.15 shows the percentage of queries returning relevant answers within various top K ranks for the two query types respectively. The x axis represents various top K ranks, ranging from 1 to 1600 in log scale. The y axis reports the percentage of the tested queries returning relevant answers under a certain rank. Evaluation method 1 is used for getting the rank of relevant answers of queries.

Table 2.3 and 2.4 give comparison of the six ranking algorithm, on Query Type I and II respectively using $\lfloor MRR \rfloor$ and $\lceil MRR \rceil$.

| Measure | *EntityRank* | L | N | G | C | W |
|---|---|---|---|---|---|---|
| $\lfloor MRR \rfloor$ | **0.648** | 0.047 | 0.037 | 0.050 | 0.266 | 0.379 |
| $\lceil MRR \rceil$ | **0.648** | 0.125 | 0.106 | 0.138 | 0.316 | 0.387 |

Table 2.3: Query Type I MRR Comparison

| Measure | *EntityRank* | L | N | G | C | W |
|---|---|---|---|---|---|---|
| $\lfloor MRR \rfloor$ | **0.659** | 0.112 | 0.451 | 0.053 | 0.573 | 0.509 |
| $\lceil MRR \rceil$ | **0.660** | 0.168 | 0.454 | 0.119 | 0.578 | 0.520 |

Table 2.4: Query Type II MRR Comparison

As we can see from all the results, our *EntityRank* algorithm is highly effective with MRR around 0.65, outperforming all the other algorithms. "Ordered Window" pattern "ow" is used in our *EntityRank* algorithm for evaluating those queries.

| | Query $C3$ | | Query $R4$ | |
|---|---|---|---|---|
| | $P$ | $R$ | $P$ | $R$ |
| Illinois | $\frac{54}{59} = .92$ | $\frac{54}{54} = 1.0$ | $\frac{45}{53} = .85$ | $\frac{45}{54} = .83$ |
| Indiana | $\frac{20}{25} = .80$ | $\frac{20}{30} = .67$ | $\frac{25}{27} = .93$ | $\frac{25}{30} = .83$ |
| Wisconsin | $\frac{36}{41} = .88$ | $\frac{36}{37} = .98$ | $\frac{31}{38} = .82$ | $\frac{31}{37} = .84$ |
| **Overall** | $\frac{110}{125} = \mathbf{.88}$ | $\frac{110}{121} = \mathbf{.91}$ | $\frac{101}{118} = \mathbf{.86}$ | $\frac{101}{121} = \mathbf{.83}$ |

Figure 2.16: Accuracy: Precision and Recall for $C3$ and $R4$

**Recall Comparison**

While it is viable to evaluate precision by manually checking the returned results, measuring recall of the returned results is a much harder task. For some queries (e.g., professors related with database research), it is very difficult, if not impossible, to get the complete set of correct results.

Our effort to measure the recall of query results specifically focuses on a small domain related with computer science departments in US, where we can readily figure out all the possible answers.

We created query $C3$ for finding $\langle$#professor, #email, #university$\rangle$ for three universities: Indiana University, Illinois, and Wisconsin. Let $D$ be our discovered relation (as a *set* of discovered tuples) and $T$ be the "truth" relation (as a set of correct tuples). As standard metrics, we use *precision* $P = \frac{|D \cap T|}{|D|}$ and *recall* $R = \frac{|D \cap T|}{|T|}$. As the "truth," We manually collected the correct tuples for all CS professors at the three universities.

As Figure 2.16 reports, we obtained an overall $P$=.88 and $R$=.91 – which is rather satisfactory as a result of automatic mining. Specifically, Illinois showed very high recall and precision, while Indiana and Wisconsin are somewhat lower. This difference can be explained, in part, due to the fact that email addresses are often disguised (for "anti-robot" extraction) on Indiana and Wisconsin pages, by not showing an @ sign or domain names. In spite of this, the accuracy numbers are still quite high, showing the promise of *ER* discovery at a large scale.

We also created query $R4$, for finding $\langle$#prof, #univ, #research$\rangle$. As the "truth" to compare to, we note that each #prof has a unique #univ but often has multiple #research areas– thus we compare to their top three areas (manually compiled). Overall, we obtained $P = .86$ and $R = .83$, as Figure 2.16 summaries in the detail. These results are particularly impressive, because the associations between all three entities must be correct.

Although the focus of this chapter is on the effectiveness of entity search, we have also carried out some preliminary study on the efficiency of our system, in terms of space and time.

First, in terms of space consideration, supporting entity search only incurs minimal index overhead. Our current entity search system, by indexing email and phone entities, only incurs less than 0.1% overall space overhead in the size of indices.

Second, in terms of time consideration, our system adds negligible overhead in offline indexing time and performs online entity search rather efficiently. Indexing entities could be done at the same time as we index keywords. Similar to the space overhead it creates, the time overhead in indexing entities in addition to keywords is almost negligible. For online query processing, as the nature of the query processing is linear as discussed in Section 2.4.4, query processing is rather efficient. For our examples queries listed in Figure 2.13 and 2.14, the average query response time is 2.45 seconds.

**Observations and Discussions**

We now analyze why the other five approaches perform not as well as our *EntityRank* algorithm.

In the local model only approach (L), as long as there is some false association where keywords and entities appear very close to each other, the tuple will be ranked high. Our global aggregation of the local scores is more robust, as the results are collective from many sources across the web and such false association is not likely to appear in lots of web pages. The results for the local model only approach may get improved by having more accurate local models, however, it doesn't solve the problem from the root as we analyzed. It is necessary to conduct global aggregation upon local analysis.

On the other extreme, using pure global aggregation without any local constraint, *e.g.* the N and G ranking methods, performs poorly as our result shows. This is because without local constraint, lots of false associations will be returned, which leads to the aggregation of false information. Local model helps in reducing such noises, generating high quality information for later on aggregation.

Experiments show that both a simple combination of the local scores with global aggregation and performing *EntityRank* without hypothesis testing perform worse than *EntityRank*. This validates our analysis on the important factors for entity search in that the lacking of any factor, in this case the discriminative and associative factors, would result in significant reduction in effectiveness.

Finally, we would like to point out an intriguing merit of *EntityRank* in that it performs holistic analysis , by leveraging the scale of the corpus, together with effective local analysis. Therefore, the larger the corpus size, the more information (evidence) we can aggregate, and therefore the more effective the ranking. Other simple methods, especially the local model only approach, may suffer from such situations as the larger the corpus the more the noise.

## 2.6 Related Work

The objective of our system is to build an effective and efficient entity-aware search framework, with a principled underlying ranking model, to better support many possible search and mining applications. Our previous work [21] formulates the problem of entity search and emphasize its application on information integration (*i.e.*, Scenario 4 in Section 2.5.2), while this work focuses on the core entity ranking problem. To the best of our knowledge, we did not witness any similar exploration of combining local analysis, global aggregation and result verification in a principled way, especially over a network of documents of varying quality. Our work is related with the existing literature in three major categories: information extraction (*i.e.*, IE), question answering (*i.e.*, QA) and emerging trend on utilizing entity and relation for various search tasks. We now study these categories one by one.

First, our system on one hand relies on IE techniques to extract entities; on the other hand, our system could be regarded as online relation extraction based on association. There have been many comprehensive IE overviews recently ( [27], [30], [4]) summarizing the state of the art. On the special Web domain, there have been many excellent IE works (*e.g.*, SemTag [34], KnowItAll [35], AVATAR [41]) Furthermore, many open source frameworks that support IE (*e.g.*, GATE [1], UIMA [2]) are readily available. While most IE techniques extract information from single documents, our system discovers the meaningful association of entities holistically over the whole collection.

Second, our system can be used as a core component to support QA more directly and efficiently. Unlike most QA works (*e.g.*, [3], [11], [48], [49]), which retrieve relevant documents first and then extract answers, our work directly builds the concept of entity into the search framework. While many QA systems' focus is on developing interesting QA system framework, most of them have adopted simple measures for ranking and lack a principled conceptual model and a systematic study of the underlying ranking problem. The SMART IR system [3] and the AskMSR QA system [11] mainly use the entity occurrence frequency for ranking. The Mulder system [48] ranks answer candidates mainly according to their closeness to keywords, strengthened by clustering similar candidates for voting. The Aranea system [49] mainly uses the frequency of answer candidates weighted by idf of keywords in the candidates as the scoring function.

Finally, we are recently witnessing an emerging research trend towards searching with entity and relationship over unstructured text collection (such as [18] and [67] advocate).

Towards enriching keyword query with more semantics, AVATAR [43] semantic search tries to interpret keyword queries for the intended entities and utilize such entities in finding documents.

There has been plenty of work on supporting search over semi-structured data such as XML documents, and structured data such as relational databases. Studies on using keyword search over XML documents,

such as [38], [39], focus on finding and ranking XML documents, while we aim at finding and ranking more fine-granularity objects: entities.

Towards searching over fully extracted entities and relationships from the Web, ExDB [15] supports expressive SQL-like query language over an extracted database of singular objects and binary predicates, of the Web; Libra [59] studies the problem of searching web objects as records with attributes. Due to the different focus on information granularity, its language retrieval model is very different from ours. While these approaches rely on effective entity and relationship extraction for populating an extraction database, our approach only assumes entity level extraction and replies on large-scale analysis in the ranking process.

Towards searching over typed entities in or related with text documents, BE [13] develops a search engine based on linguistic phrase patterns and utilizes a special index for efficient processing. It lacks overall system support for general entity search with a principled ranking model. Its special index, "neighborhood index", and query language, "interleaved phrase" query, are limited to phrase queries only; Chakrabarti et al. [19] introduce a class of text proximity queries and study scoring function and index structure optimization for such queries. Its scoring function primarily uses local proximity information, whereas we investigate effective global aggregation and validation methods, which we believe are indispensable for robust and effective ranking in addition to local analysis. Our query primitive is also more flexible in allowing expressive patterns and multiple entities in one query; ObjectFinder [17] views an "object" as the collection of documents that are related with it and therefore scores an "object" by performing aggregation over document scores. In contrast, our approach views an entity tuple as all its occurrences over the collection. Therefore, its score aggregates over all its occurrences, where we consider uncertain, contextual factors other than the document score. In addition, while their focus is more on the query efficiency side, by exploiting Top-K queries using early termination approach, our work is primarily on the query effectiveness side.

# Chapter 3

# Efficient, Scalable Entity Search with Dual-Inversion Index

## 3.1 Introduction

The immense scale and widespread of the Web has rendered it as our ultimate repository and enriched it with all kinds of *data*. With the diversity and abundance of "things" on the Web, we are often looking for various information objects, much beyond the conventional *page view* of the Web as a corpus of HTML pages, or documents. The Web is now a collection of *data objects*, where pages are simply their "containers." The page view has inherently confined our search to reach our targets "indirectly" through the containers, and to look at each container "individually."

With the pressing needs to exploit the rich data, we have witnessed several recent trends towards finding fine granularity information *directly* and across many pages *holistically*. This chapter attempts to distill these emerging search requirements, abstract the function of underlying search, and develop efficient computation for its query processing. Such requirements arise in several areas:

**Web-based Question Answering (WQA)** Question answering has moved much towards Web-based: Many recent efforts (*e.g.*, [11, 49, 70]) exploited the diversity of the Web to find answers for ad-hoc questions, and leverage the abundance to find answers by simple statistical measures (instead of complex language analysis). As requirements, to answer a question (*e.g.*, "where is the Louvre Museum located?"), WQA needs to find information of certain type (a location) near some keywords ("louvre museum"), and examine as many evidences (say, counting mentions) to determine the final answers.

**Web-based Information Extraction (WIE)** Information extraction, with the aim to identify information systematically, has also naturally turned to Web-based, for harvesting the numerous "facts" online—*e.g.*, to find *all* the ⟨museum, location⟩ pairs (say, ⟨Louvre, Paris⟩). Similar to WQA, Web-based IE exploits the abundance for its extraction: correct tuples will appear in certain patterns more often than others, as testified by the effectiveness of several recent WIE efforts (*e.g.*, [35, 13, 54]). As requirements, WIE thus needs to match text with contextual patterns (*e.g.*, order and proximity of terms) and aggregate matching across many pages.

**Type-Annotated Search (TAS).** As the Web hosts all sorts of data, as motivated earlier, several efforts (*e.g.*, [19, 13, 24]) proposed to target search at specific *type* of information, such as *person* names near "invent" and "television." As requirements, such TAS, with varying degrees of sophistication, generally needs to match some proximity patterns between keywords and typed terms and to combine individual matchings into an overall ranking.

We believe these emerging trends all consistently call for, as their requirements agree, a non-traditional form of search—which we refer to as *entity search* [24]. Such search targets at various typed unit of information, or *entities*, unlike conventional search finding only pages. In this chapter, we use #-prefixed terms to refer to entities of a certain type, *e.g.*, #location or #person. Observing from WQA, WIE, and TAS, we note the unanimous requirements following the change of targets from pages to typed entities:

- *Context matching*: Unlike documents which are searched by keywords in its *content*, we now match the target type (say #location) by keywords (*e.g.*, "louvre museum") that appear in its surrounding *context*, in certain desired patterns (*e.g.*, within 10 words apart and in order).
- *Global aggregation*: Unlike documents which appear only once, we match an entity (say, #location = Paris) for as many times as it appears in numerous pages, which requires us to globally aggregate overall scores.

While the requirements for entity search have emerged, we have not tackled the computational challenges for efficiently processing such queries. Recent works have focused on effective scoring models mostly (*e.g.*, [19, 24]). For query processing, a widely adopted form (as in many WQA works [11, 49, 70]) is to "build upon" page search—to first find matching pages by keywords, and then scan each page for matching entities. This "baseline" (Sec. 3.3), much like *sequential scan*, is hard to scale, and thus it may work only by limiting to top-$k$ pages— which will impair ranking effectiveness (Sec. 3.2).

As the main theme of this chapter, for efficient and scalable entity search, we must index *entities* as first-class citizen, and we identify the "dual-inversion" principle for such indexing. We recognize the concept of *inversion* from the widely-used inverted lists. To index entities, we thus parallel the standard keyword-to-document inversion in dual perspectives: From the *input* view, we see entity as keyword, from which we develop "document-inverted" index. From the *output* view, we see entity as document, from which we derive "entity-inverted" index. The dual-inversions can coexist, and form the core of our solution.

For parallel query processing upon such indexes, we see the challenge in the interplay of join and aggregate: By viewing entity indexes as relations, we capture entity search as, in nature, an *aggregate-after-join* query—a particular type of groupby-join query that is hard to parallelize. Intuitively, the needs for context matching lead to *complex join* between relations, while global aggregation leads to *group by and aggregate*. We design data partition and query processing for the dual-inversion framework.

Finally, we evaluate our methods over a real Web crawl of 150 million pages (3 TB), with a diverse set of 21 entity types. To be realistic, we designed two concrete application scenarios ("Yellowpage" and "CSAcademia"), which together have 176 queries in four benchmark sets. Our experiments reveal that both types of inversions can dramatically speed up entity search—with "entity-inverted" at *2-4 orders* of magnitude difference and "document-inverted" at *1-3 orders*. The space overhead of indexing is quite acceptable: "document-inverted" tends to slightly increase index size from standard keyword indexing, while "entity-inverted" implies reasonable space overhead (and sometimes can even result in smaller size based on different domains). Overall, this chapter makes the following contributions:

- We distill and abstract the essential *computation requirements* for entity search.
- We systematically derive and propose novel *dual-inversion indexing and partition* schemes for efficient and scalable query processing.
- We verify our design over a *realistic, large-scale Web corpus* with concrete applications.

## 3.2 Abstraction & Challenges

Towards designing a framework for entity search, we start with characterizing its functions and challenges.

**Functional Abstraction.** An entity search system provides search over a set of supported entity types $\{E_1, \ldots, E_n\}$, which we informally consider as the *schema*. *E.g.*, our CSAcademia application in Sec. 3.5 has schema (#university, #professor, ...). Each type $E_i$ is a set of *entity instances* that are pre-extracted from the corpus (*e.g.*, "201-7575" $\in$ #phone). As the requirements indicate (Sec. 3.1), we abstract entity search as follows:

*Entity Search (ES) Problem:* Give a document collection $D$, for a query $\alpha(k_1, \ldots, k_m, E)$ with keywords $k_i$ (*e.g.*, "database systems") and entity type $E$ (*e.g.*, #professor), ES will find *entity instances* $e \in E$ and rank them by $score(e)$ which matches context pattern $\alpha$ and aggregates all matching occurrences across $D$.

To illustrate, from our prototype (Sec. 3.5), Fig. 3.1 shows the screenshot for query $Q_{db}$ "database systems #professor" (with default $\alpha$ as "order, 20-word window" written as ow20), for the first 5 results and supporting pages (where each answer appears). Notice, typically top results are supported by more than 1 page, as the ranking relies on aggregation. Fig. 3.1 shows one support page for each result just for conciseness.

We observe that, functionally, entity search (ES) is a generalization of page search (PS) in several ways:

- Entity as first class citizen: Unlike PS assuming page as *the* entity, ES can support any recognizable entity.

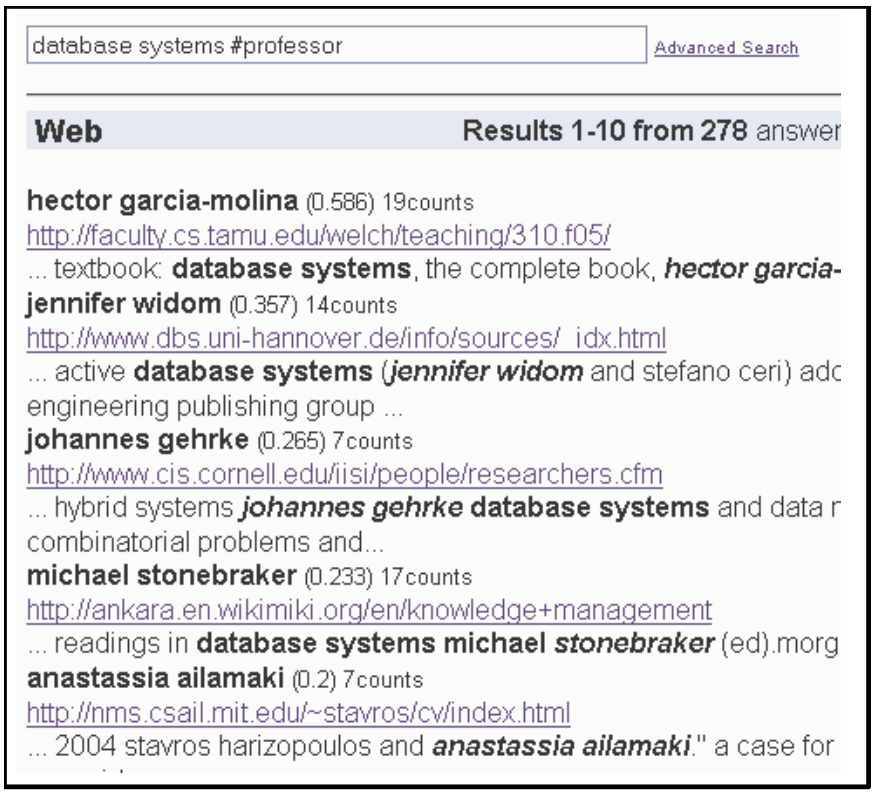



Figure 3.1: Result: "database systems #prof"

- Set as output: Unlike PS targeting at only a few *top* relevant results, an ES query can generally require a *set* of answers; *e.g.*, the above example (Fig. 3.1) can return tens or hundreds of relevant professors.
- Holistic aggregate: Unlike PS assuming each page as "unique," ES must generally handle entities occurring multiple times. Finding and returning such supporting evidences is crucial for applications, such as WQA, to actually determine the correct answers.

**Computational Requirements.** The objective of ES, as just abstracted, is to rank entity $e$ (as instance of the target type $E$) by a scoring function $score(e)$. The choice of scoring function will directly impact the quality (or "relevance") of the ranked results. However, as this chapter focuses on the computation framework, we will identify the key components of such ranking functions (Sec. 3.3 will give example scoring functions). Our previous work [24] addresses the quality of search results.

As the requirements of ES, as Sec. 3.1 identifies, a reasonable scoring function should capture both *context matching* and *global aggregation*. Consider scoring an entity $e$. For our discussion, let $o\langle \mathsf{doc}, \mathsf{pos}\rangle$ denote an *occurrence* of $e$ in some page doc at word position pos (recall that an entity instance can occur many times in corpus $D$). Similarly, we use $\kappa_j\langle \mathsf{doc}, \mathsf{pos}\rangle$ as an occurrence of keyword $k_j$.

1. *Context matching:* The first step in scoring is to match the occurrences of $k_i$ and $e$ to the desired context pattern $\alpha$. We assume a *local matching function* $L_\alpha$. Given occurrences $\kappa_i$ for keywords $k_i$ and $o$ for entity

$e$, $L_\alpha$ will assess how well the positions match $\alpha$ by some similarity function $\text{sim}(\cdot)$, if the occurrences are in the same page.

$$L_\alpha(\kappa_1, \ldots, \kappa_m, o) = \begin{cases} 0, \text{if } \kappa_i.\mathsf{doc} \text{ and } o.\mathsf{doc} \text{ differ;} \\ \text{sim}(\alpha, \kappa_1.\mathsf{pos}, \ldots, o.\mathsf{pos}), \text{else.} \end{cases} \tag{3.1}$$

2. *Global aggregation:* The second step is to aggregate all the occurrences across pages. Here some function $G$ aggregates the local scores globally into the total score, across all occurrences $o$ in $D$.

Thus, to summarize, the essential computation to calculate the score, $score(e)$, is generally of the form:

$$score(e) = G_{(\kappa_1,\ldots,\kappa_m,o)\in d, d\in\mathcal{D}}[L_\alpha(\kappa_1, \ldots, \kappa_m, o)], \tag{3.2}$$

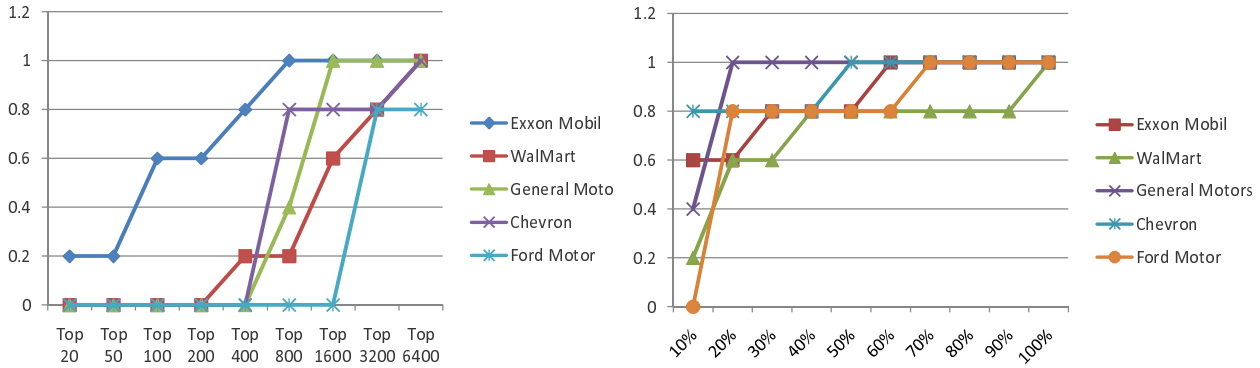
**Challenges.** Document search has often relied on a small number of high quality documents for pruning, and therefore avoiding the need to scan full inverted lists (*e.g.*, [51]) for high efficiency. Such pruning techniques, however, are not directly applicable to entity search, with the mandate on processing comprehensive corpus due to the following two major reasons:

First, since entity search relies on *global aggregation*, comprehensive corpus is needed to generate stable aggregative statistics. Second, many entity queries are naturally looking for *set output* of comprehensive results over the entire corpus (*e.g.*, "#professor in DB" as in $Q_{db}$ or "#city in California"). Figure 3.2(a) and 3.2(b) show the accuracy (measuring top 5 results) of 5 typical queries of finding the phone number of companies (represented by the $y$ axis), by varying the number and percentage of top documents (returned by issuing keyword queries against a document search engine) used respectively (represented by the $x$ axis). Evidently, different queries converge to accuracy 100% at very *different* points, indicating that it is nontrivial to determine which "top k" value to stop for different queries. Moreover, queries generally require a significant portion (over 40%) of all the relevant documents for stable results.

Given these two points, this work assumes processing query over the entirety of the corpus, without considering pruning. We believe studying approximate query answering by performing intelligent dynamic pruning is itself an interesting research problem. However, such study is beyond the scope of this work.

Overall, we thus recognize two essential challenges in building an efficient framework, which goes much beyond traditional document search:

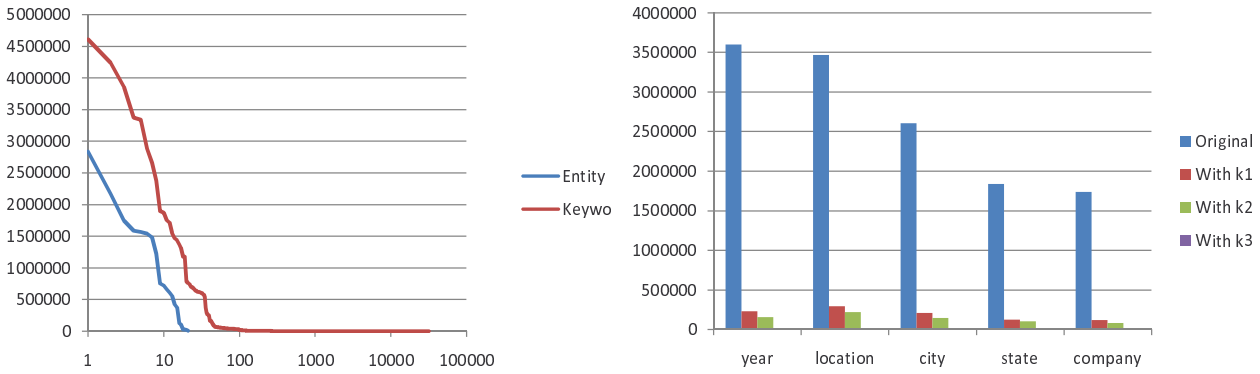
**Complex Join:** As we see from the problem abstraction of entity search, each query involves at least one entity. Unlike keywords, entities, comprised of many entity instances, tend to appear frequently across the




(a) Absolute Page Count (b) Page Percentage Count

Figure 3.2: Top K Comparison for Point Queries

entire corpus. Figure 3.3(a) shows the comparison of keyword frequency (*i.e.*, the number of times a keyword appears in corpus) with entity frequency (*i.e.*, the number of times an entity type, say #phone, appears in corpus), with $x$ axis in log scale representing keywords/entities under comparison, and $y$ axis representing their respective frequencies. As seen from the figure, entities clearly appear much more frequently (by orders of magnitude) than most of the keywords, with frequency comparable to the top 20 most frequent keywords. Therefore, it is computationally expensive to load/check those many occurrences of entities for pattern matching. In addition, as discussed in the characteristics of entity search, it has to rely on in-document *contextual pattern matching*. Such computation is also more expensive, compared to the traditional simple document intersection checking in document search.




(a) Frequency Comparison (b) Selectivity of Keywords

Figure 3.3: Keyword and Entity

On the other hand, we also see potential opportunities to reduce computation. With respect to a specific query, only a small fraction of the entity occurrences are actually related to the query, due to the selectivity of keywords. Figure 3.3(b) shows the frequency of entity alone, as compare to the frequency of entity when combined with keywords. 5 random entity types together with 3 random keywords are used in the experiment. As we can see, given a keyword, most of the entity instance occurrences are irrelevant. This

observation opens up room for expediting processing, which we will exploit further in our solution.

**Global Aggregation:** As we discussed in the characteristics in Sec. 3.1, entity search has to rely on *holistic aggregation* over comprehensive corpus to tap the rich redundancy of the Web. This is an online processing layer that is non-existent in traditional document search. Being able to support aggregating information over large-scale corpus in an online fashion is another essential computation requirement for entity search. How can we parallelize such online large-scale aggregation for scaling up?

The challenge of this work is thus to deal with the essential computation requirements of entity search, towards an efficient and scalable framework to support entity search.

## 3.3 Baseline & Running Example

To set the stage of discussion, let's use Fig. 3.4 as a running example throughout the chapter, which we call the *YellowPage* scenario, as it provides search for contact information (*e.g.*, #phone, #email). As a toy dataset, the corpus $\mathcal{D}$ has 100 documents $\mathcal{D} = \{d_1, \ldots, d_{100}\}$; we show three documents $d_6$, $d_9$, $d_{97}$ as examples. We will assume a simple query for finding the phone number of Amazon service $Q1$ in the following form:

$Q1$: ow20(amazon service #phone)

During offline processing, we recognize the position of keywords (*e.g.*, keyword "amazon" appears at position 17 of document $d_6$) in the corpus (via tokenization). Entities are also extracted offline, with their entity instances identified and positions recognized. *E.g.*, we extract phone number 800-201-7575 as phone instance $p_8$ at position 19 of document $d_6$ as shown in Fig. 3.4. Notice, there may be additional properties related with the extracted entity occurrences, *e.g.*, the extraction confidence. We exclude such information in this chapter for the ease of discussion.

For our concrete discussion, let's assume a simplistic scoring function, *BinarySum*, in our running example.

**Example 1.** *[Scoring Function: BinarySum] Let's define scoring scheme BinarySum, which instantiates Eq. 3.2 by:*

$$\begin{aligned} L_\alpha(e) &= 1, \textit{if } e \textit{ matches } \alpha; 0 \textit{ otherwise.} \\ G &= Sum \end{aligned}$$

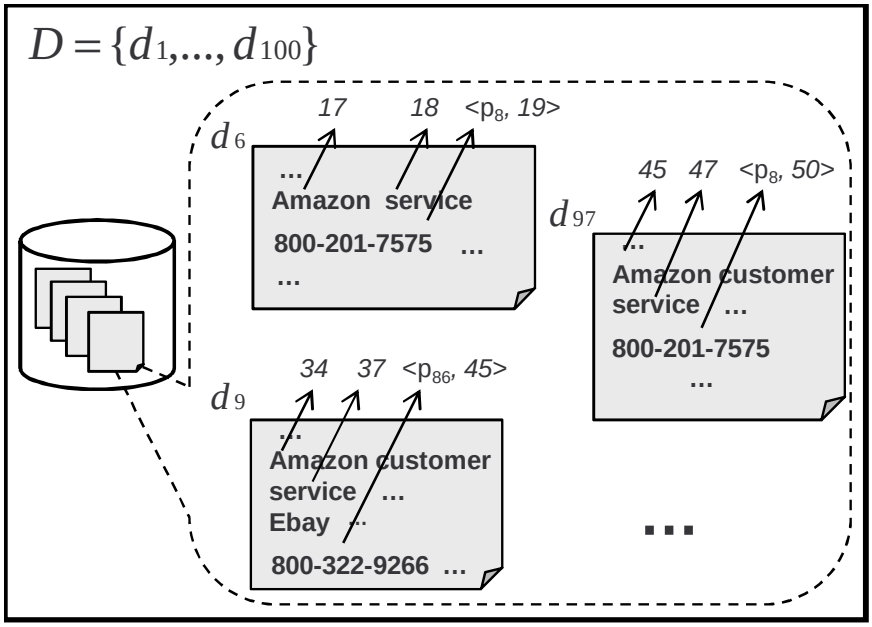



Figure 3.4: A Running Example: *YellowPage*

*These definitions lead to a rather simplistic scheme, which scores entity instance e by simply counting the total number of times it occurs in a way matching the α-pattern. While BinarySum may not be effective in ranking, it is sufficient as a concrete example for discussing the essential computation.*

To execute the general form of Eq. 3.2, as an entity-centric system is currently lacking, many related works (*e.g.*, [3, 11, 48, 49, 70]) have relied on keyword-based search engines to zoom into a subset of documents and then apply local matching by scanning documents and global analysis. Considering example $Q1$, the *baseline* goes as follows:

1. Look up by keywords (*e.g.*, "amazon service" for $Q1$) in a keyword-based search engine for retrieving documents matching these keywords. From the running example in Fig. 3.4, documents $d_6$, $d_9$ and $d_{97}$ will be returned as matching documents.
2. Scan each matching document to execute local matching $L_\alpha$ to match candidate entities. Notice, as keywords and entities are all identified offline, only pattern matching (by pattern $\alpha$) needs to be performed. In the example for instance, $p_8$ will be matched from document $d_6$ with local matching score of 1 by *BinarySum*.
3. Perform aggregation to assemble the produced matchings. In the example, $p_8$ will have an aggregate score of 2 from $d_6$ and $d_{97}$ using *BinarySum*.

In order to handle large-scale dataset, it is common practice to partition the corpus into sub-corpses, and distribute the sub-corpses. Over the baseline, step 1&2 can be processed over the partitioned sub-corpse in

parallel, while step 3 aggregates the results generated from the sub-corpses.

Our discussion will assume a parallel setting of "$p + 1$" nodes with two processing layers, with $p$ nodes assigned for storing indexes and local processing, and 1 node assigned for global processing. We choose this "$p + 1$" setting to focus on indexing, partition, and parallel processing over the $p$ local nodes. The global processing layer, which can also be parallelized using multiple nodes, is simplified to one node.

The keyword-based baseline, while not meeting the requirement of entity search as it needs to perform expensive document scan, and rely on central aggregation, has been popularly used for QA tasks over the Web–The lack of efficiency and scalability, and the popularity nonetheless, indicate a clear demand for a true entity search system.

## 3.4 Solutions: Dual-Inversion Index

We now develop the solutions for supporting entity search. Our key issue, as just motivated is—How to design an index to facilitate query processing? In this section, we will reason the design to derive two types of indexes that work well together—which we call the "dual-inversion" index.

To begin with, we recognize that, for text retrieval, the key principle of indexing is *inversion*– an efficient data structure for mapping from query *input values* to *output objects*. In a standard text search scenario, users give *keywords* as input values and expect *documents* as output objects; *i.e.*, we are searching in a database of documents by keywords. Thus, the standard inversion that powers up today's text retrieval is mapping from keywords, as input values, to document as candidates for output objects. Since text databases are not optimized for real-time updates, an index does not need to be "dynamic," (unlike database indexes such as B-tree) and thus the most efficient data structure is simply a sequential list of such mappings, called *inverted list*—one list for each keyword—where each *posting* is one document ID and the positions in the document where $k$ occurs. Such lists can be efficiently loaded from disk into memory by sequential read, or compressed and cached in memory [72].

We can express this standard inversion—mapping a keyword $k$ to a document collection $D$—as the following (one to many) mapping from $k$ to those documents in $D$ whose content contains $k$, as follows:

$$D(k) : k \ \rightarrow \ \{\langle \mathsf{doc}, \mathsf{pos}\rangle |\ \mathsf{doc.content}[\mathsf{pos}] = k; d \in D\}. \tag{3.3}$$

We will develop our indexing based on the principle of inversion. Thus, our question becomes, what inversions shall we develop to support entity search? Why?

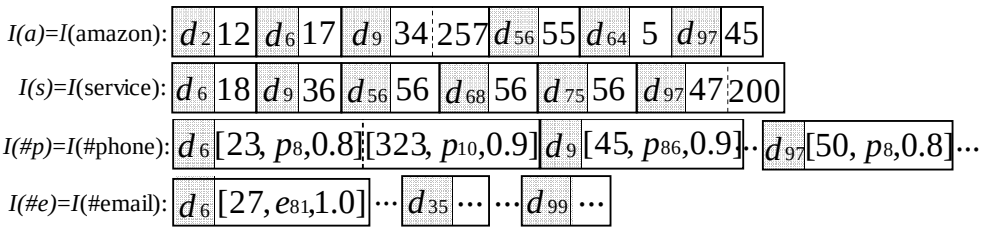



Figure 3.5: Document-Inverted Index Example

### 3.4.1 Document-Inverted Index

The first proposal naturally parallels keyword inversion $D(k)$: Just like keywords for document search, entity type serves as input for entity search.

**Indexing: Document Inverted.** In the functional form, entity search takes as input both keywords and entity types: Given an entity-search query $\alpha(k_1, \ldots, k_m, E)$, since the entity type $E$ is part of the input, just like keywords $k_i$, can we build a mapping for $E$ in the same way as $k_i$—because they are both *input*? We consider as the first inversion $D(E)$ which, given entity type $E$, maps to the documents where entity of type $E$ occurs. As the target of inversion is documents, we refer to this scheme as *document-inverted* index, or *D-Inverted* index for short.

To realize this analogous concept, however, there is a slight complication: Unlike keywords which are literal, $E$ is an "abstract" type– which can have different instance values. Thus the mapping should record, in addition to document $d$ and position $p$, the specific entity instance $\mathsf{entity}$ of type $E$, for each occurrence.

$$D(E) : E \rightarrow \{\langle \mathsf{doc}, \mathsf{pos}, \mathsf{entity}\rangle |\ d.\mathsf{content}[\mathsf{pos}] = \mathsf{entity}; \mathsf{entity} \in E; d \in D\}. \tag{3.4}$$

Fig. 3.5 shows the layout of the inverted lists $D(a)$, $D(s)$ and $D(\#p)$ for keywords "amazon", "service", and entity type "#phone" respectively.

For further development for query processing, we can conceptualize each inverted list as a *relation*: As Exp. 3.3 and 3.4 show, each list is simply a set of postings of the same structure—or "tuples"—and thus $D(k)$ is a relation with schema $\langle \mathsf{doc}, \mathsf{pos}\rangle$ and $D(E)$ a relation with schema $\langle \mathsf{doc}, \mathsf{pos}, \mathsf{entity}\rangle$. Note that the relational view is conceptual, allowing us to understand the operations, and we do not necessarily use a DBMS to store and process the lists.

**Computation Analysis.** With this document-based inversion in place, we now capture the computation for query processing. Given the document-inverted lists $D(k_i)$ and $D(E)$, how do we process them to answer

the query $\alpha(k_1, \ldots, k_m, E)$? We are starting from the D-Inverted lists as base relations $D(k_1), \ldots, D(k_m)$ and $D(E)$.

Specifically, we can now use relational operations to describe the essential operations. Starting from the base relations, our objective is to score every entity instance $e$ by Eq. 3.2 and sort all the instances by their scores. First, to find all the qualifying entity occurrences, we perform join between the relations $D(k_1), \ldots, D(k_m)$ and $D(E)$. We call such join *context join* as it evaluates a context pattern $\alpha$ over the occurrences of $k_1, \ldots, k_m$, and some entity occurrence of $E$. It checks whether the occurrences match the pattern $\alpha$ and scores how well a matching is by the local scoring function $L_\alpha$—thus, strictly speaking, it is a "fuzzy" join that returns scores. Second, we need a groupby operator $\mathcal{G}$ to group entity occurrences according to their instances, *i.e.*, $D(E)$.entity, and use global aggregation function $G$ to calculate the final score for each instance. Finally, a sort operator $\mathcal{S}_{score}$ sorts the entity instances. We show the overall computation in Exp. 3.5.

$$\mathcal{S}_{score}[_{(D(E).\mathsf{entity})}\mathcal{G}G(\ltimes_{L_\alpha}[D(k_1), \ldots, D(k_m), D(E)])] \tag{3.5}$$

Written in SQL, in this view, entity search is to execute the following query $Q_{ES1}$.

SELECT $D(E)$.entity, $G$(mscore) AS *score*
FROM $D(k_1), \ldots, D(k_m), D(E)$
WHERE $L_\alpha(D(k_1)$.doc, $D(k_1)$.pos, ...,
$D(k_m)$.doc, $D(k_m)$.pos, $D(E)$.doc, $D(E)$.pos) AS mscore
GROUP BY $D(E)$.entity
ORDER BY *score* ($Q_{ES1}$)

The query is an instance of *aggregate-join* query, which has the following general form in SQL:

SELECT $R_1.G_1$, ..., $R_n.G_n$, $Agg(R_1.A_1$, ..., $R_n.A_n)$
FROM $R_1$, ..., $R_n$
WHERE $Join(R_1.J_1$, ..., $R_n.J_n)$
GROUP BY $R_1.G_1$, ..., $R_n.G_n$
HAVING/ORDER BY ...

Such aggregate-join queries connect tuples from base relations and organize them into groups for aggregation, *i.e.*, with the following two parts:

- **Join:** It joins relations $R_1$, ..., $R_n$, through an expression, denoted $Join$, of join conditions (which include selections), upon *join attributes* $J_1$, ..., $J_n$. Each $J_i$ can be an attribute or multiple attributes of $R_i$.
- **Group-By:** It then groups the joined tuples over *group-by attributes* $G_1$, ..., $G_n$, and then aggregates each group with some function $Agg$ over aggregate attributes $A_1$, ..., $A_n$.

Such aggregate-join queries impose particular issues in parallel query processing—which arise in our specific situation. To explain and contrast the issues, let's use the following query $Q_{bank}$ over a typical "bank" scenario. Consider two relations Customers(cid, name, address) and Accounts(accountno, cid, branch, balance). Query $Q_{bank}$ finds those customers having more than $50000 as total balance across all their accounts.

```
SELECT C.name, Sum(A.balance) as TotalBal
FROM Customers C, Accounts A
WHERE C.cid = A.cid
GROUP BY C.cid
HAVING TotalBal > 50000                    (Q_bank)
```

As the query form involve both join and aggregate, can we push group-by and aggregate to be performed before join? While such transformation is desired, to reduce the expensive join cost, and possible in some cases, it is not feasible in our scenario. For instance, consider $Q_{bank}$; suppose Accounts has 10000 tuples but only 100 distinct cid values. We can perform Group-By on Accounts first to result in 100 cid-groups, and perform Having over the groups. This transformation will reduce the number of Accounts tuples to join with Customers from 10000 to only the less-than-100 cid-groups after grouping and filtering. Unfortunately, this transformation is not possible for entity search. Consider $Q_{ES1}$. Observe that the global scoring function $G$ requires mscore as computed by the local scoring function $L_\alpha$ for every tuple combination from $D(k_1), \ldots, D(k_m), D(E)$—by comparing their doc and pos to match pattern $\alpha$. That is, the overall aggregate function $G \circ L_\alpha$ (composition of $G$ and $L_\alpha$) needs aggregate attributes from *all* the relations, unlike $Q_{bank}$ only needs balance from Accounts. Thus, for $Q_{bank}$, Group-By (and aggregate) must happen after join, or we do not have all the aggregate attributes. While we explain intuitively, a full analysis of the feasibility of transformations is discussed in [71].

**Data Partition: Document Space.** To process entity search queries, now that we need to process aggregate after join, how to partition the relations for parallel query processing? To scale up entity search over a large corpus, we must partition data somehow over the $p$ worker nodes. Our particular form of

aggregate-join query is tricky for parallelization, because the join and group-by are over a different set of attributes—*i.e.*, in terms of the general form, $J_i \neq G_i$. To contrast, for $Q_{bank}$, since both the join and aggregate are over attribute cid, we can simply partition Customers and Accounts by the same *cid* ranges, and each worker node can execute both join and aggregate.

Unfortunately, when join and group-by are over different attributes, as in our situation, no schemes can fully partition the corpus for both join and aggregate without significant replication of communication. Naturally, we can partition on either join or aggregate attributes, as observed in [63, 64]. We next discuss these choices:

As the first choice, we may partition relations by their group-by attributes, which turned out to be infeasible for entity search. Referred to as *APM* [63], this aggregation partition method will partition each relation $R_i$ by $G_i$. If $R_i$ does not appear as part of Group-By (*i.e.*, $G_i = \emptyset$), then the entire $R_i$ needs to be broadcast to all the nodes at run time (or otherwise every $R_i$ needs to be replicated to every node). For entity search, as *offline data partitioning*, we partition $D(E)$ by $D(E).\mathsf{entity}$ into sub-relations, $D^1(E)$, ..., $D^p(E)$, for the $p$ local worker nodes; *i.e.*, records of the same entity instance will distribute to the same node. At *runtime processing*, for query $\alpha(k_1, \ldots, k_m, E)$:

1. Broadcast $D(k_1)$, ..., $D(k_m)$ to every local node.
2. Each local node $z$ will join $D^z(E)$ with $D(k_1)$, ..., $D(k_m)$, group-by entity, aggregate for each group, and send the results to the global node.

$$_{(D^z(E).\mathsf{entity})}\mathcal{G}_G(\ltimes_{L\alpha}\ [D(k_1), \ldots, D(k_m), D^z(E)]) \tag{3.6}$$

3. The global node unions and sorts all the $p$ result sets, to produce the overall ranking of the entity instances.

Clearly, this scheme is infeasible, with the run time cost to broadcast the inverted lists of the queried keywords to worker nodes (Step 1). Or, we may simply replicate every keyword lists, *i.e.*, $D(k)$ for every possible $k$ to each node. Given the numerous keywords possible in any corpus, replication is again prohibitive. Thus, aggregate-based partition will not work.

As the other choice, thus, we will partition by the join attributes. Referred to as *JPM* (join partition method) in [63], this method will partition each relation $R_i$ by $J_i$. For entity search $Q_{ES1}$, we are matching pattern $\alpha$ by the context-join $\ltimes_{L_\alpha}$ over the keyword and entity relations on their doc and pos attributes. To determine the partition, we must examine— What are the conditions that these tuples from each relation are "joinable"—*i.e.*, $L_\alpha(D(k_1).\mathsf{doc}, D(k_1).\mathsf{pos}, \ldots, D(k_m).\mathsf{doc}, D(k_m).\mathsf{pos}, D(E).\mathsf{doc}, D(E).\mathsf{pos}) > 0$? Since

we are matching entities and keywords from each document, any joinable occurrences must be at least from the same document. More formally, by the definition of $L_\alpha$ as Eq. 3.1 gives, the context join between $D(k_1)$, ..., $D(k_m)$, $D(E)$ must require that

$$D(k_1).\mathsf{doc} = \cdots = D(k_m).\mathsf{doc} = D(E).\mathsf{doc}.$$

Thus, with the principle of join-based partition, we will partition the D-Inverted relations by the *document space*—*i.e.*, to distribute the tuples of $D(k)$ and $D(E)$ by the document IDs they are from, or their doc attributes. We will apply this partitioning to every base relation: $D(k)\langle\mathsf{doc}, \mathsf{pos}\rangle$ and $D(E)\langle\mathsf{doc}, \mathsf{pos}, \mathsf{entity}\rangle$, for all keywords $k$ and for all entity types $E$ supported by the system. For each relation, we will distribute the postings with the same doc to the same local nodes—As discussed above, these are postings that are "joinable." Specifically, first, we partition the "document space" $D$ into $p$ disjoint subsets—one for each local node—*i.e.*, $D^1$, ..., $D^p$, such that $D^1 \cup \ldots \cup D^p = D$ and $D^i \cap D^j = \emptyset$. With respect to the $p$ document sub-spaces, we then distribute each D-Inverted index to the $p$ local nodes, as follows:

$$\begin{aligned} D^z(k) &= \{x | x \in D(k), x.\mathsf{doc} \in D^z\} \\ D^z(E) &= \{x | x \in D(E), x.\mathsf{doc} \in D^z\} \end{aligned}$$

Each local node will host the corresponding sublist for each keyword, and entity. For instance, the document-inverted index of entity #phone in Fig. 3.5 will be split into $p$ sublists, and the $i$-th sublist will be located on the $i$-th local node. For the *YellowPage* scenario, assuming we have 10 local processing units, we can partition the dataset containing 100 document into 10 subset, each containing 10 documents as shown in Fig. 3.6. This implies the inverted index will be partitioned into sublists. For instance, $D(a)$ in Fig. 3.5 will be partitioned into 10 sublists, $D^1(a)$, ..., $D^{10}(a)$ respectively.

**Parallel Query Processing.** The local processing module, having all the information of a subset of the documents, will be able to compute all the *context join*s and output all the matching entity occurrences. Exp. 3.7 formulates this procedure, where the matching entity occurrences are put into $L^z$ and will be sent over to the global processing module for further processing. This step implements the context join operation in Exp. 3.5 in parallel across local nodes.

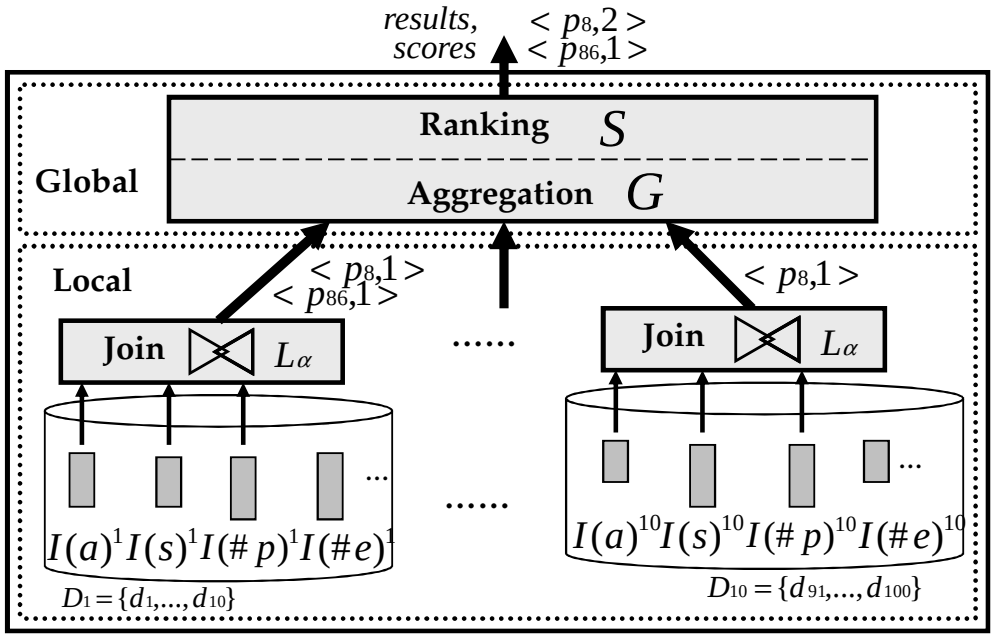



Figure 3.6: Partition by Document Space

**Local Node $z$:** : $\forall z \in [1..p]$

$$L^z\langle \mathsf{entity}, \mathsf{mscore}\rangle = \pi_{\mathsf{entity},\mathsf{mscore}} \ \bowtie_{L_\alpha \text{ as } \mathsf{mscore}} [D^z(k_1), \ldots, D^z(k_m), D^z(E)] \tag{3.7}$$

The local matching algorithm D-Local in Fig. 3.7 loads the document-inverted index for the specified keywords, and entity into memory (step 2). As the lists are sorted based on document id, merge-join can be performed over the lists to instantiate any possible matchings (step 3-10). If a matching entity occurrence is found, we will use local scoring function $L$ to compute the score, and output (step 5-8).

To answer query $Q_1$, we will execute the query on each of the local nodes as shown in Fig. 3.6. Local node 1 will produce two matchings for phone instance $p_8$ matched in document $d_6$ and $p_{86}$ matched in document $d_9$ by joining sublists $D^1(a)$, $D^1(s)$ and $D^1(\#p)$. Local node 10 will produce one matching for phone instance $p_8$ matched in document $d_{97}$.

With the entity occurrences and local scores produced, we are ready to perform *holistic* aggregation over them. The global processing module takes care of both aggregation and sorting over all the matching entity occurrences in $L^1, \ldots, L^p$ collected from all the local nodes, as shown in Exp. 3.8:

**Algorithm D-Local:**
*Local* Processing with *D-inverted* Index, for Node $z$.

- **Input:** Query $\alpha(k_1, \ldots, k_m, E)$.
- **Output:** $L^z\langle entity, mscore\rangle$.

```
1: L^z = ∅
2: load lists D^z(k_1), ..., D^z(k_m), D^z(E)
3: for merge-join over the loaded lists do
4:    if D^z(k_1).doc = ... = D^z(E).doc then
5:       if D^z(k_1).pos, ..., D^z(E).pos match α then
6:          mscore = L_α (D^z(k_1).pos, ..., D^z(E).pos)
7:          add (D^z(E).entity, mscore) to L^z
8:       end if
9:    end if
10: end for
11: return L^z
```

Figure 3.7: Algorithm D-Local

**Global Node:** :

$$\mathcal{S}_{\mathsf{score}}\big[_{(\mathsf{entity})}\mathcal{G}_{G(\mathsf{mscore})\ \text{as}\ \mathsf{score}}(L^1 \cup \cdots \cup L^p)\big] \tag{3.8}$$

The global aggregation algorithm D-Global in Fig. 3.8 goes through all the input matched entity occurrences, and aggregates all the scores of a specific instance together. As shown in Fig. 3.6, the global processing layer receives the matching occurrences from the local nodes, and performs aggregation and ranking. For instance, the local scores for $p_8$ are aggregated into the final score of 2, resulting the ranking of $p_8$ at the first place.

### 3.4.2 Entity-Inverted Indexing

Our second proposal parallels keyword inversion in an "opposite" way. While our first inversion, D-Inverted indexing, views entity type $E$ as input and maps it to documents, we now consider entities as output— the target of search.

**Indexing: Entity Inverted** In the functional form, entity search finds entity instances as output from keywords as input: Given query $\alpha(k_1, \ldots, k_m, E)$, we are looking for entities $e$ of type $E$, such that keywords

**Algorithm D-Global:**
*Global* Processing with *D-inverted* Index.

- **Input:** $L^z\langle entity, mscore\rangle$, $\forall z \in [1..p]$.
- **Output:** ranked list of $\langle entity, score\rangle$.

```
1: Result = ∅
2: for each ⟨entity, mscore⟩ in L^1, ..., L^p do
3:    if entity not in Result then
4:       add entity to Result
5:    end if
6:    update Result[entity].score with mscore by G
7: end for
8: sort Result by score; return Result
```

Figure 3.8: Algorithm D-Global

$k_i$ appear in the *context* of $e$ in a way that matches pattern $\alpha$. *E.g.*, in our example, we are given "amazon service" to search for entity #phone that are mentioned with these keywords around it (in that sequential pattern).

With this view, we again seek to parallel the traditional inversion. We observe that traditional document search builds upon keyword inversion $D(k)$, as Exp. 3.3 shows, which maps each keyword $k$ as query input to documents in $D$ as output. For entity search, we shall map each keyword $k$ to entities $\in E$, denoted $E(k)$. As the inversion targets to entities, we call $E(k)$ an *entity-inverted* index, or *E-Inverted* index for short.

To realize this analogous concept, however, we again face some interesting complications— While a document only occurs once (or we do not capture duplicates in document search), each entity can occur multiple times in the text corpus at different documents or different positions. Thus, while building E-Inverted index, as the target of mapping, we must specify to the level of a specific *occurrence*, rather than just an entity instance. To specify an occurrence, denoted $o$, we will specify the *document* and *position* where an *entity* occurs—thus the tuple $o\langle$doc, epos, entity$\rangle$. With this notation, we build an E-Inverted index for each keyword $k$ by mapping $k$ to the context of some entity occurrence $o$ where $k$ appears. Each "posting" record will be of the form $\langle o\langle$doc, epos, entity$\rangle$, pos$\rangle$, which means $k$ appears, with position pos in the context of entity occurrence $o\langle$doc, epos, entity$\rangle$.

$$E(k) : k \ \rightarrow \ \{\langle o\langle \mathsf{doc}, \mathsf{epos}, \mathsf{entity}\rangle, \mathsf{pos}\rangle |\ \mathsf{o.context[pos]} = k; \mathsf{entity} \in E\}. \tag{3.9}$$

As the second issue, we also must define what *context* means—*i.e.*, how far from an entity occurrence shall we consider as *within* its context? We note that, for our first document-based inversion, the *content* of a document is well defined. Here, to define the "context" of an entity occurrence $o$, we are essentially considering the question—How far apart between $k$ and $o$ do we consider them as no longer "semantically associated"? Clearly, larger the distance is, the less likely they are associated, and most entity-oriented search efforts (*e.g.*, [19, 24]) leverage this insight in ranking. Thus, in our indexing, we can choose some maximal window distance to consider as context. In our implementation, we use 200-word window as the context—*i.e.*, the context of an entity occurrence extends between 100 words to its left and 100 to its right. Fig. 3.9 shows the layout of the entity-inverted index using our example.

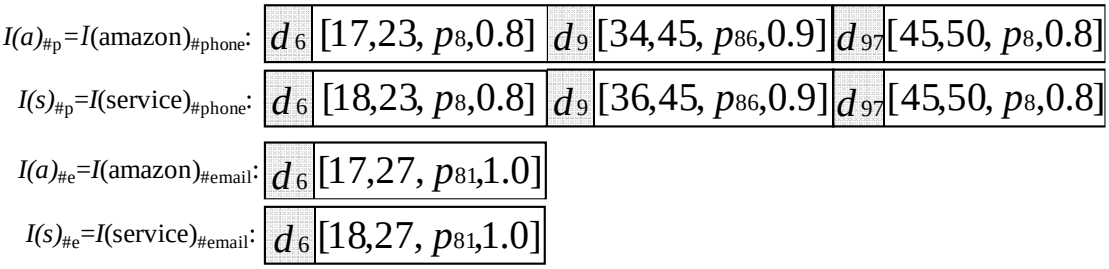



Figure 3.9: Entity-Inverted Index Example

Thus, with entity-inverted indexing, as we store the mapping of keywords to entities, we have as base relations the entity-inverted lists $E(k)$ with schema $\langle$doc, epos, entity, pos$\rangle$.

**Computation Analysis.**

Starting from these base relations, in contrast to Exp. 3.5, we can express the computation of entity search for $\alpha(k_1, \ldots, k_m, E)$ as:

$$\mathcal{S}_{score}[_{(D(k_1).\text{entity})}\mathcal{G}_G(\bowtie_{L_\alpha}[E(k_1), \ldots, E(k_m)])] \tag{3.10}$$

Written in SQL, in this view, entity search is to execute the following query $Q_{ES2}$.

SELECT $E(k_1)$.entity, $G$(mscore) AS score
FROM $E(k_1), \ldots, E(k_m)$
WHERE $L_\alpha(E(k_1)$.doc, $E(k_1)$.epos, $E(k_1)$.entity, $E(k_1)$.pos, ...,
$E(k_m)$.doc, $E(k_m)$.epos, $E(k_m)$.entity, $E(k_m)$.pos)
AS mscore

GROUP BY $E(k_1)$.entity

ORDER BY score $(Q_{ES2})$

We have $Q_{ES2}$, again, as an instance of *aggregate-join* query. *First*, like $Q_{ES1}$, the query must also handle aggregate after join— The overall aggregate function $G \circ L_\alpha$ needs aggregate attributes from *all* the relations to get pos attributes for matching $\alpha$. *Second*, however, unlike $Q_{ES1}$, this query based on entity-inversion relations has the same attributes—the entity attributes of each relation—for both aggregate and join.

With this key difference, the entity-inversion view allows us to simultaneously parallelize both join and aggregate, since now join and aggregate attributes are consistent.

**Data Partition: Entity Space.**

To partition along the entity groups, we make sure the same instances of $E$ will be allocated at the same local node, which means we must divide $E$ into disjoint subsets. Specifically, we partition $E$ to $p$ nodes, *i.e.*, $E = \cup(E^1, \ldots, E^p)$ and $E^i \cap E^j = \emptyset$. With respect to the $p$ entity sub-spaces, we can distribute each E-Inverted index to the $p$ local nodes, as follows:

$$E^z(k) \quad = \quad \{x = \langle o, \mathsf{pos} \rangle | x \in E(k), o.\mathsf{entity} \in E^z\}$$

Again in our example setting, using the same 10 local processing units, we could partition dataset as shown in Fig. 3.10 such that local node 1 is responsible for phone entity instances $p_1, \ldots, p_{10}$. Take the list $E(a)$ in Fig. 3.9 as an example. This list will be split into two nonempty sublists. Local node 1 will hold sublist $E^1(a)$ with entries $d_6 : [23, p_8, 17]$ and $d_{97} : [50, p_8, 45]$ and local node 9 will hold sublist $E^9(a)$ with entry $d_9 : [45, p_{86}, 34]$.

**Parallel Query Processing.** Upon the entity space partition scheme, the local processing module can perform the joining operation, as well as the aggregation operation. In other words, Exp. 3.10 can be fully realized at each local node (except for the final ranking part):

**Local Node $z$:** : $\forall z \in [1..p]$

$$L^z\langle \mathsf{entity}, \mathsf{score} \rangle = \pi_{\mathsf{entity},\mathsf{score}\ (E^z(k_1).\mathsf{entity})}\mathcal{G}_{G(\mathsf{mscore})\ \text{as}\ \mathsf{score}}(\bowtie_{L_\alpha\ \text{as}\ \mathsf{mscore}}[E^z(k_1), \ldots, E^z(k_m)]) \tag{3.11}$$

We illustrate the local matching&aggregation algorithm in Algorithm E-Local in Fig. 3.11. It loads the entity-inverted index for the specified keywords with regard to the input entity (step 2). As the lists are

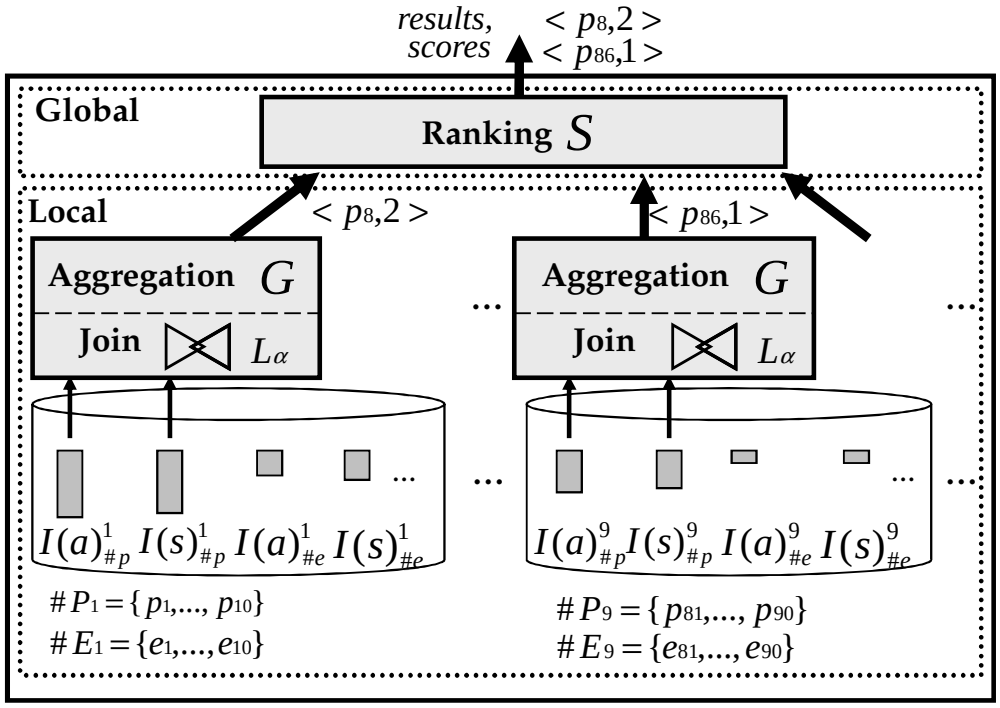



Figure 3.10: Partition by Entity Space

**Algorithm E-Local:**
*Local* Processing with *E-inverted* Index, for Node $z$.

- **Input:** Query $\alpha(k_1, \ldots, k_m, E)$.
- **Output:** $L^z\langle entity, score\rangle$.

```
1:  L^z = ∅
2:  load lists E^z(k_1), ..., E^z(k_m)
3:  for merge-join over the loaded lists do
4:    if E^z(k_1).o = ... = E^z(k_m).o then
5:      let o be the entity occurrence in common
6:      if E^z(k_1).pos, ..., E^z(k_m).pos, o.epos match α then
7:        mscore = L_α (E^z(k_1).pos, ..., E^z(k_m).pos, o.epos)
8:        if o.entity not in L^z then
9:          add o.entity to L^z; initialize score to 0
10:       end if
11:       update entity's score with mscore by G
12:     end if
13:   end if
14: end for
15: return L^z
```

Figure 3.11: Algorithm E-Local

**Algorithm E-Global:**
*Global* Processing with *E-inverted* Index.

- **Input:** $L^z\langle entity, score\rangle, \forall z \in [1..p]$.
- **Output:** ranked list of $\langle entity, score\rangle$.

1: $Result = L^1 \cup \cdots L^p$
2: sort *Result* by *score*
3: return *Result*

Figure 3.12: Algorithm E-Global

sorted based on document id, merge-join can be performed over the lists to instantiate any possible matchings (step 3-14). If a matching entity occurrence is found, we will use local scoring function $L$ to compute the score (step 6-7). This score will be immediately aggregated with the produced occurrences (step 8-11).

To answer the same query, the query will be issued on each local node as shown in Fig. 3.10. As the entity-inverted index is still ordered by document id, the same sort-merge join algorithm can be applied. In this setting, the two matchings of phone instance $p_8$ will both be produced from local node 1 by joining sublists $E^1(a)$ and $E^1(s)$. Unlike in the document partition based approach, these matching can already be grouped and aggregated on the local nodes. In this example, the final query score of phone instance $p_8$ is calculated on node 1 and that of $p_{86}$ is calculated on node 9.

Given that the local processing module produces aggregated results, the global processing module only has to take care of the ranking step in Exp. 3.10 of all the aggregated results from $L^1, \ldots, L^p$, a very light-weight task as shown in Exp. 3.12 and algorithm E-Global in Fig. 3.12:

$$\underline{\textbf{Global Node:}}: \quad \mathcal{S}_{\mathsf{score}}[L^1 \cup \cdots \cup L^p] \tag{3.12}$$

### 3.4.3 Together: Dual-Inversion Index

We summarize the pros and cons of D-Inverted and E-Inverted proposals in terms of the computation requirements we listed in Sec. 3.2, pattern join, aggregation, as well as the space requirement, in the following table:

**Pattern Matching:** Baseline is slow as it performs pattern matching by scanning documents returned from keyword search. D-Inverted and E-Inverted schemes are fast in utilizing indexes for efficient pattern matching. However, the E-Inverted scheme is more efficient, as it deals with much shorter index lists, whereas

| | Baseline | D-Inverted | E-Inverted |
|---|---|---|---|
| Pattern Join | slow | fast | faster |
| Aggregation | central | central | distributive |
| Space | standard | minimal overhead | large |

Table 3.1: Comparison of the Two Indexes

the D-Inverted scheme has to load and read long D-Inverted lists for entities.

**Aggregation:** E-Inverted scheme allows the aggregation to be fully distributed in parallel to local nodes. The baseline and the document-inverted index schemes, on the other hand, have to rely on a central layer for aggregation.

**Space:** The space overhead for the D-Inverted scheme is rather minimal, as it only creates one D-Inverted list per entity. The entity-inverted index scheme could often incur more significant space cost, as we combine entity with every keyword.

As the two schemes are highly complementary to each other, we ask: can the two types of index coexist to reach a nice balance point? Fortunately, the two types of indexes can indeed coexist, as each contains complete information with respect to the entity. This offers us the opportunity to create entity-inverted index for a selected set of entity types, while the rest of the entity types can be supported by document-inverted index. Generally, entity-inverted index should be created for entities that are queried more often and take less space, whereas document-inverted index should be created for the rest of entities which are queried less frequently and require more space. We name such a framework, with the coexistence of the two types of indexes, the *dual-inversion index* framework.

## 3.5 Experiments

To empirically evaluate our dual-inversion approaches for entity search, for its efficiency over a large scale corpus and diverse types of entities, in a range of realistic benchmark scenarios, we built a distributed prototype on a real Web corpus of a 3TB general Web crawl (collected in January 2008) with 150 million pages. Like the "$p$+1" setting described in Sec. 3.3, we ran the system on a cluster of 15 local worker nodes ($p$=15) and one global node, totally 16 machines, each with a dual AMD Athlon 64 X2 3600+ CPU, 1 GB memory and 1TB of disk.

On this large corpus, we annotated a wide range of various entity types—21 entities total—in order to understand different application scenarios. We used the GATE system [1] for entity annotation. As Table 3.2 lists, we selected our entities covering the three major extraction methods: using dictionaries, rules, and

classifiers (machine learning).

| Method | Supported Entities |
|---|---|
| dictionary-based | **14 entities:** Country, City, State, Province, Region, Sea, Company, Title, Drug, Month, University, ResearchArea, Professor, Religion |
| rule-based | **4 entities:** Email, Phone, Zipcode, Year |
| classifier-based | **3 entities:** Person, Location, Organization |

Table 3.2: Supported Entity Types: 21 Entities

For our comparison, we implemented all the three approaches discussed: the keyword-based Baseline (Sec. 3.3) and the dual-inversion: D-Inverted and E-Inverted index (Sec. 4.3). As Table 3.3 summarizes, all the three methods, including Baseline, had the entities pre-extracted offline. As indexes, the Baseline used standard keyword inverted lists $D(k)$, and D-Inverted added $D(E)$ in addition, while E-Inverted used only keyword-to-entity inversion $E(k)$. (We will compare the space requirements later.) All the methods are parallelized across the same $(p+1)$-node cluster, by partitioning the index data as we discussed.

| Method | Extraction | Indexes Built |
|---|---|---|
| *Baseline* | offline | $D(k)$, $\forall$ keyword $k$ |
| *D-Inverted* | offline | $D(k)$, $\forall$ keyword $k$; $D(E)$, $\forall$ entity $E$ |
| *E-Inverted* | offline | $E(k)$, $\forall$ keyword $k$ |

Table 3.3: Indexes Built for Each Method

**Experiment Setup.** To extensively and realistically study the performance, we configured two concrete applications. We evaluated 4 benchmark sets, for totally 176 queries of varying parameters. Each query has the form $\alpha(k_1, \ldots, k_m, E)$, as Sec. 3.2 defines, for keywords $k_i$ and entity type $E$. We use "ow20" for pattern $\alpha$— ordered 20-word window— for all queries. As scoring function, we use the "EntityRank" model [24], which is of the common form of $G \circ L_\alpha$ as Sec. 3.2 defines. We stress that the actual function affects "only" ranking preciseness. For our focus of efficiency, all functions with the join-then-aggregate ($L_\alpha$ then $G$) abstraction are computationally similar.

**Application 1 (Yellowpage)** for finding yellowpage-like information, with entities (#email, #phone, #state, #location, #zipcode).

• Benchmark 1A *Phone Number Search*: **30 queries** of the form "company name #phone", *e.g.*, "general motors #phone", which finds the phone number related to General Motors. We generated 30 queries using top 30 company names in 2006 Fortune 500.

• Benchmark 1B *Location Search*: **20 queries** of the form "city #location", *e.g.*, "springfield #location", which finds locations related with Springfield. We generated 20 queries using Illinois city names.

**Application 2 (CSAcademia)** for information of the computer science academia, with entities (#university, #professor, #research, #email, #phone).

• Benchmark 2A *Email Search*: **88 queries** of the form "researcher #email", *e.g.*, 'Anastassia Ailamaki #email", which finds emails related to the researcher. We generated 88 queries using PC members of SIGMOD 2007.

• Benchmark 2B *Professor Search*: **38 queries** of the form "area #professor", *e.g.*, "database systems #professor", which finds professors related to the area. We generated 38 queries using CS areas like data mining, compiler, *etc.*.

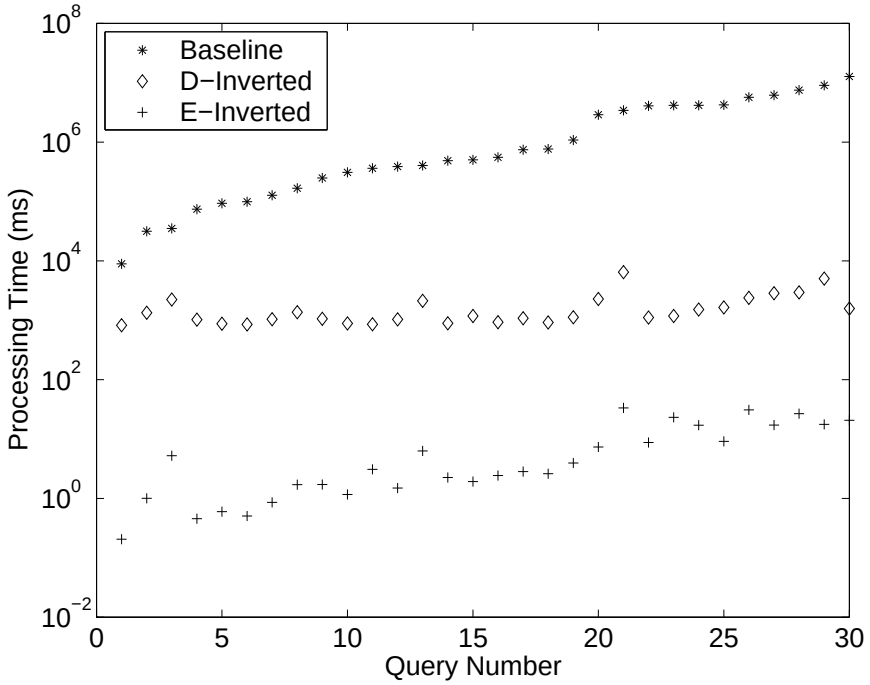
We chose these benchmark queries not only because of their practical usefulness but also their diversity: First, they contain both set answers (1B, 2B) and single points (1A, 2A). Second, they differ in the selectivity of keywords. Benchmark 1A and 1B have keywords (*e.g.*, "IBM", "Chicago", *etc.*) that are far less selective than 2A and 2B (*e.g.*, "Ailamaki", "HCI"). Third, they cover entities extracted with different methods (Table 3.2).

While we focus on efficiency, we note that the usefulness of entity search is also revealing through these benchmarks. *E.g.*, As Sec. 3.2 mentioned, Fig. 3.1 shows the screenshot for "database systems" #professor with supporting pages. Such queries, with page search, would require us to comb through numerous page results to collect answers. Entity search expands our ability to directly find fine grained information holistically across many pages.

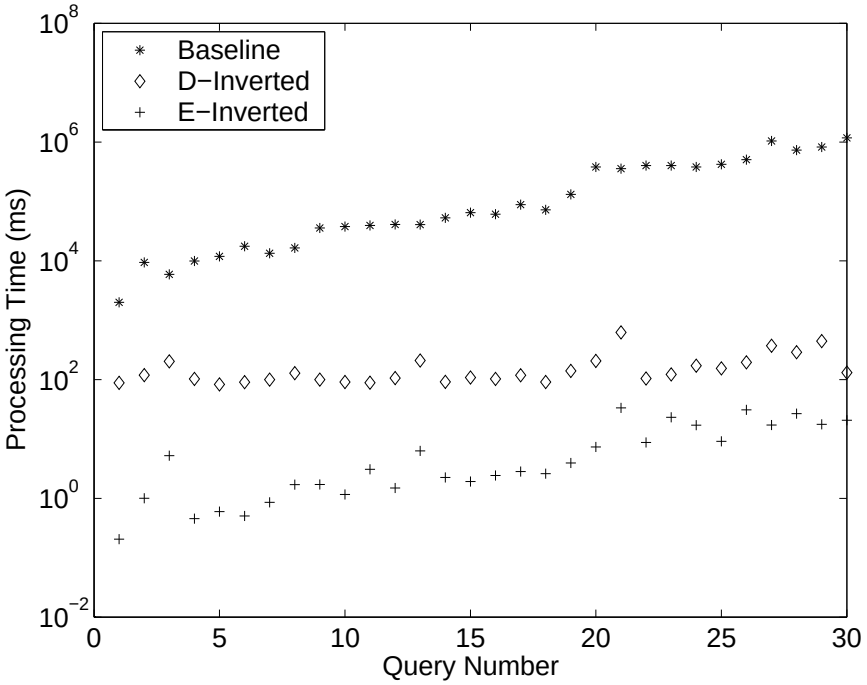
**Performance Evaluation.** We focus on search efficiency, and evaluate each component: processing at the $p$ local nodes, network transfer, and processing at the global node, with the following metrics. $M1$: overall local processing time. $M2$ : max local processing time. $M3$: overall transfer time. $M4$: max transfer time. $M5$: global processing time. When involving local nodes, we measure both *overall* as the sum of all nodes (which indicates *throughput*), and *max* as the maximum (which indicates *response* time).

Fig. 3.13 shows the local times for 1A (queries are sorted by overall local processing time in Baseline). Both D-Inverted and E-Inverted incur much less overall and max local processing time than Baseline, and E-Inverted performs faster than the D-Inverted. As the graphs are in log scale, we observe rather significant speedup–generally *two* orders of magnitude: E-Inverted ranges around $10^2 ms$, D-Inverted $10^4 ms$, and Baseline $10^6 ms$. Furthermore, the times for D-Inverted and E-Inverted are more uniform than the Baseline, which has high variance in the number of documents needed to scan after keyword lookup.

Fig. 3.14 shows the transfer times for 1A. Notice, the cost for Baseline and D-Inverted are the same (thus the points collapsed together), since they send the same partial "after-join" results to the global node.



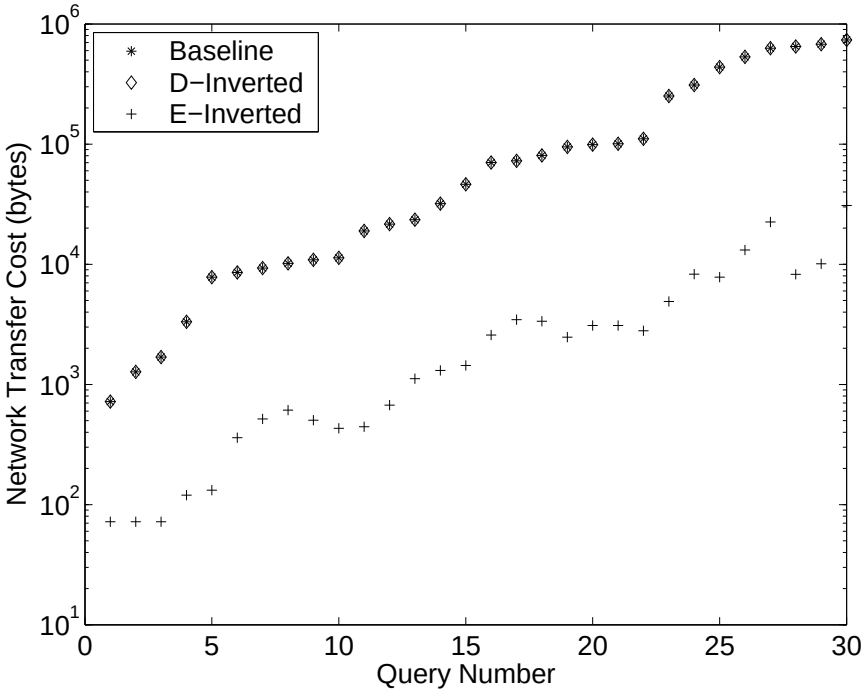

(a) M1: Overall Local Processing.



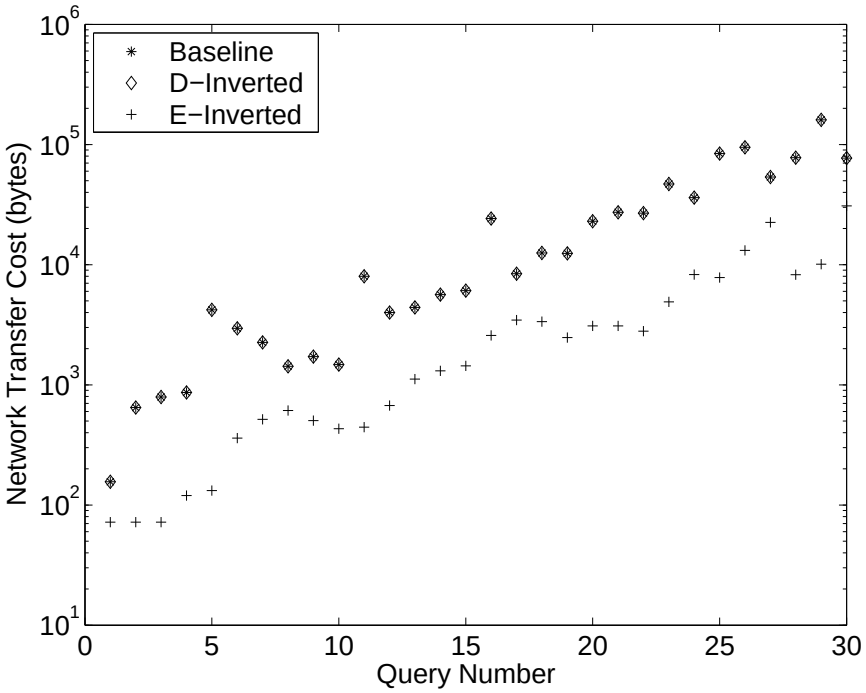

(b) M2: Max Local Processing.

Figure 3.13: Local Processing: Benchmark 1A

We observe that E-Inverted can save significantly in network transfer cost, as results are already "after-aggregation." The difference is, again, significant–at about *two* orders of magnitude. Notice, in the case of only outputting top-k results, E-Inverted scheme can further save transfer cost, as at most top-k results




(a) M3: Overall Network Transfer.




(b) M4: Max Network Transfer.

Figure 3.14: Network Transfer: Benchmark 1A

from each local node need to be sent for final ranking.

Fig. 3.15 shows the global times for 1A. E-Inverted requires much less global processing time compared

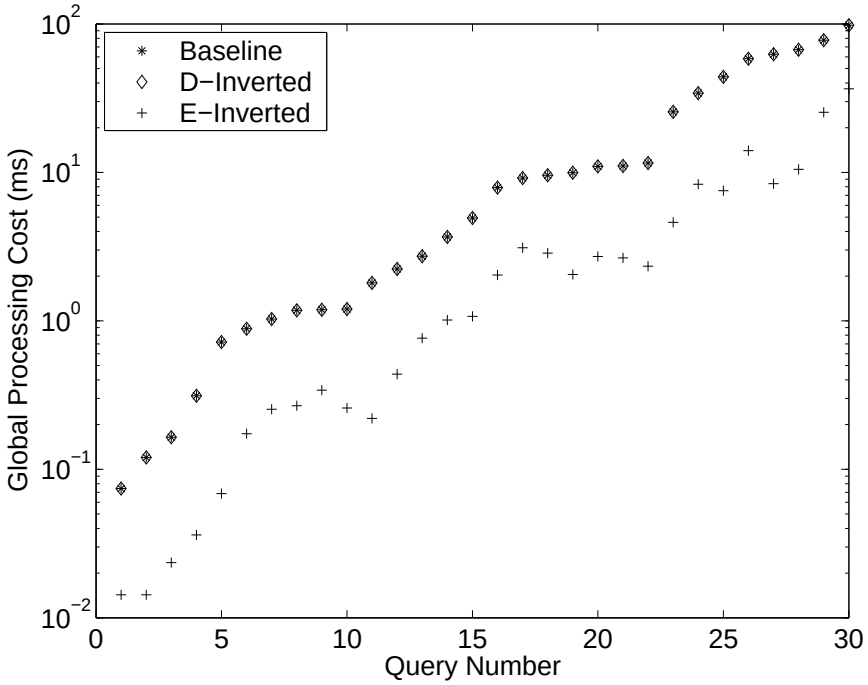



Figure 3.15: Global Processing (M5): Benchmark 1A

with the Baseline and D-Inverted (which have the same global costs). The difference is about *one* order of magnitude.

Overall, we observe similar results for all the benchmarks, 1A, 1B, 2A, and 2B, in both applications. Table 3.4 summarizes the *median* cost of all the $M1$ to $M5$ metrics. We consistently observe, from Table 3.4, across the four benchmarks of totally 176 queries, the significant speedup of the dual-inversion approaches, for all the processing components $M1$ to $M5$. Both E-Inverted and D-Inverted are much faster than the Baseline– which use keyword indexes to look up pages for entity search.

We conclude by comparing the time efficiency and space overhead for our dual-inversion approaches. Table 3.5 summarizes the average (across all the queries in each benchmark set) total execution times for all the three methods. To compare the dual-inversions to Baseline, we also compute the *speedup* for each category in the parentheses; *e.g.*, for benchmark 1A (30 queries), E-Inverted has an average speedup of 2.5E+4 or $2.5 \cdot 10^4$. Across the categories, we see rather significant speedup from 1 to 4 orders of magnitude.

The speedup comes at the cost of indexing entities—recall index configuration in Table 3.3. Table 3.5 also compares the various index sizes of the two application settings. *First*, we observe that, since D-Inverted relies on $D(E)$ in addition to standard keywords $D(k)$, it always requires larger index size than Baseline– However, the addition is actually quite small, resulting in 1% and 0.1% size increase in Application 1 and 2, respectively. *Second*, we observe that, with its entity-primary indexing on $E(k)$, E-Inverted can require varying indexing sizes, depending on the actual entities indexed. In Application 1, E-Inverted requires 89.7%

| | Metric in Median | Baseline | D-Inverted | E-Inverted |
|---|---|---|---|---|
| 1A | M1 (s) | 527.5 | 1.14 | **0.04** |
| | M2 (s) | 62.8 | 0.119 | **0.003** |
| | M3 (kb) | 58 | 58 | **24.4** |
| | M4 (kb) | 8.2 | 8.2 | **1.95** |
| | M5 (ms) | 6.41 | 6.41 | **1.55** |
| 1B | M1 (s) | 6075 | 25.44 | **2.23** |
| | M2 (s) | 570.5 | 3.26 | **0.44** |
| | M3 (kb) | 5687 | 5687 | **127** |
| | M4 (kb) | 648 | 648 | **9.5** |
| | M5 (ms) | 579.43 | 579.43 | **98.24** |
| 2A | M1 (s) | 46.5 | 1.13 | **0.01** |
| | M2 (s) | 12 | 0.096 | **0.002** |
| | M3 (kb) | 0.558 | 0.558 | **0.306** |
| | M4 (kb) | 0.144 | 0.144 | **0.036** |
| | M5 (ms) | 0.047 | 0.047 | **0.0003** |
| 2B | M1 (s) | 61 | 1.14 | **0.002** |
| | M2 (s) | 12 | 0.1 | **0.0002** |
| | M3 (kb) | 0.732 | 0.732 | **0.336** |
| | M4 (kb) | 0.144 | 0.144 | **0.036** |
| | M5 (ms) | 0.06 | 0.06 | **0.0003** |

Table 3.4: Summary of Metrics

more space, while it actually save space in Application 2 with a reduction of 80.7% index size. The variation comes from varying *selectivity* of an entity: Some entities are very frequent, such as #location in Application 1, which result in long entity-inverted indexes. Other more "specialized" entities are much less frequent, such as #university in Application 2.

| | | Baseline | D-Inverted | E-Inverted |
|---|---|---|---|---|
| Average Time (sec) | 1A | 245.61 | 0.16 (1.5E+3) | 0.01 (2.5E+4) |
| | 1B | 1348.20 | 3.88 (3.4E+2) | 2.21 (6.1E+2) |
| | 2A | 3.14 | 0.11 (2.9E+1) | 0.01 (3.1E+2) |
| | 2B | 2.03 | 0.12 (1.7E+1) | 0.01 (2.0E+2) |
| Space (TB) | App 1 | 1.45 | 1.47 (101.0%) | 2.75 (189.7%) |
| | App 2 | 1.45 | 1.46 (100.1%) | 0.28 (19.3%) |

Table 3.5: Overall: Time and Space

Overall, the experiments conclude that both types of inversions can significantly speed up entity search, while keeping space overhead acceptable. The dual-inversions, D-Inverted and E-Inverted, also present interesting tradeoff: D-Inverted generally requires minimal space addition, while E-Inverted constantly achieve higher speedup. As Sec. 4.3 discussed, both types of inversion can coexist, to balance the tradeoff– *E.g.*, in a system supporting both Application 1 and 2, we may use D-Inverted for Application 1 and E-Inverted for Application 2, resulting in small space overhead and large speedup.

## 3.6 Related Work

We are now witnessing an emerging research trend on using entities and relationships to facilitate various search and mining tasks [18, 19, 67, 46, 43, 13, 15, 17, 59, 70, 24, 73].

Our work is most related with the works on indexing unstructured documents. Inverted index [74] has been widely used in search engines for answering keywords queries. Although it is general and can support many different query types, it is not optimized for queries such as phrase queries, proximity based queries, *etc.*. Cho [26] builds a multigram index over a corpus to support fast regular expression matching. A multigram index is essentially building a posting list for selective multigrams. It can help to narrow down the matching scope. It is not optimized for phrase or proximity queries and still require full scan of candidate documents. Nextword index [69] is a structure designed to speed up phrase queries and to enable some amount of phrase browsing. It does not consider more flexible proximity based queries and does not consider types other than keywords. Indexing keyword pairs to speed up document search is studied in [52]. Our motivation to speed up entity search is different from their goal and therefore the frameworks also differ. Our index design considers entities beyond keywords, where we introduce the unique entity space partition scheme. BE [13] develops a search engine based on linguistic phrase patterns and utilizes a special "neighborhood index" for efficient processing. Although BE considers indexing types such as noun phrases other than keywords, its index is limited to phrase queries only. Chakrabarti et al. [19] introduce a class of text proximity queries and study scoring function and index structure optimization for such queries. Their study on index design is more on reducing the redundancy and the index is used for performing local proximity analysis without considering global aggregation and multi-node parallelization. Comparing with our own work [73] on supporting content querying with the design of content query language (CQL), this work focuses on the principles and foundation for the index design for facilitating efficient entity search. Moreover, this work also studies distributive computation with parallelization schemes.

There are many existing optimization techniques in IR, such as caching ([55, 52]), pruning ([60, 51]), *etc.*, to improve the efficiency of document search. Such techniques are either orthogonal to our problem, *e.g.*, caching, or can not be directly applied in our setting which requires processing over comprehensive corpus as we discussed in Sec. 3.2, *e.g.*, pruning. It is the unique computation requirements of entity search, which distinguish it from document search, that motivate us to develop novel solutions.

Since our entity search query can be viewed as "aggregate-join query" from the DB perspective, our work is also naturally related with DB literature on handling such queries ([71, 63, 64]). Such techniques are mainly designed for a small number of relations under DB setting. Our work innovates upon these works in a rather different setting: an IR setting of inverted indexes where there are almost uncountable number of

keywords.

A recent work [31] studies the indexing problem on dataspace. While this work also tries to exploit the relationship between keyword and structure, its angle from dataspace is very different from that of ours. Therefore, its index design is also very different from our schemes.

# Chapter 4

# Entity Synonym Discovery

## 4.1 Introduction

Recently, Web search has evolved into an advanced answering mechanism returning relevant facts or content, instead of just links of web pages. For example, a query such as '*Indy 4 near San Fran*', when posed on a major search engine like Bing, produces results for showtimes for the movie '*Indiana Jones and the Kingdom of the Crystal Skull*' near the city of 'San Francisco'. Using only free text to answer such queries can be problematic. On the other hand, structured data sources (*e.g.*, movie databases) often contain appropriate information for this purpose.

Effective usage of such structured data sources requires a fast and accurate match between the various query parts and the underlying structured data. However, there is often a gap between what end users type and how content creators describe the actual data values of the underlying structured entities. Content creators tend to use high-quality and formal descriptions of entities, whereas end users prefer a short, popular, and informal 'synonymous' representation. For example, a movie database lists the full title of '*Indiana Jones and the Kingdom of the Crystal Skull*', whereas Web users may type '*Indy 4*'. This phenomenon exists across virtually all domains. Apple's '*Mac OS X*' is also known as '*Leopard*'. The digital camera '*Canon EOS 350D*' is also referred as '*Digital Rebel XT*'.

Existing approaches are not always successful in automatically finding such synonymous strings. Dictionary based approaches, such as Thesaurus or WordNet [58], are insufficient when looking at the semantic alterations necessary for movies, products or company names. Substring matching based approaches work well for some cases ('*Madagascar 2*' from '*Madagascar: Escape 2 Africa*'), fall short in others ('*Escape Africa*' would also be considered incorrectly for '*Madagascar: Escape 2 Africa*') and are hopeless for the rest ('*Canon EOS 350D*' with '*Digital Rebel XT*'). Manual effort based approaches, such as Wikipedia redirect or disambiguation pages, can be of high quality, but are rather limited to only very popular entries, as we will show in Section 4.4.

The same gap between users and content creators exists in typical Web search. There it is alleviated by

the efforts of some content creators, who resort to including known alternative forms within the content of a Web page so as to facilitate a textual match by a search engine. End users resort to trying different queries until they find some Web page that satisfies them. Due to the scale of the Web, there is enough Web page content produced that when considered in unison can handle most of the different query variations.

In this chapter, we propose a fully automated solution that can enrich structured data with synonymous or alternative strings. To achieve this goal we leverage the collective wisdom generated by Web page content creators and end users towards closing the above-mentioned gap. At a high level, we use a multi-step data driven approach that relies on query and click logs to capture the Web wisdom. We first retrieve relevant Web page urls that would be good surrogates or representatives of the entity. We then identify the union of all queries that have accessed at least one of these urls by following the edges of a url-query click graph. We qualify which queries are more likely to be true synonyms by inspecting click patterns and click volume on a large subset of such urls.

We formulate the synonym finding problem in Section 4.2, and present our approach in Section 4.3. In Section 4.4, we perform a comprehensive experimental study to validate the proposed method on large-scale real-life data sets.

## 4.2 Problem Definition

In this section, we will first give our formal definitions of synonym, hypernym, and hyponym, before defining the synonym finding problem.

### 4.2.1 Synonym, Hypernym, and Hyponym

Let $\mathcal{E}$ be the set of entities over which the synonyms are to be defined. *An entity is an object with distinct and separate existence from other objects of the same type (those having similar attributes).* For example, "Indiana Jones and the Kingdom of the Crystal Skull" is an entity of type *Movie*.

Let $\mathcal{S}$ be the universal set of strings, where each string is a sequence of one or more words. We assume that there exists an oracle function $\mathcal{F}(s, \mathcal{E}) \rightarrow E$, which is an ideal mapping from any string $s \in \mathcal{S}$ that users may think of in order to refer to the very subset of entities $E \subseteq \mathcal{E}$.

We now put forward our definitions of synonym, hypernym, and hyponym in the context of entities of a specific domain.

**Definition 1 (Synonym).** *A string $s_1 \in \mathcal{S}$ is a **synonym** of another string $s_2 \in \mathcal{S}$ over the set of entities $\mathcal{E}$ if and only if $\mathcal{F}(s_1, \mathcal{E}) = \mathcal{F}(s_2, \mathcal{E})$. For example, $t_1$ = "Indiana Jones IV" is a synonym of $t_2$ = "Indiana Jones 4" since they both cover the same set of entities in the movie domain.*

**Definition 2 (Hypernym).** *A string $s_1 \in \mathcal{S}$ is a **hypernym** of another string $s_2 \in \mathcal{S}$ over the set of entities $\mathcal{E}$ if and only if $\mathcal{F}(s_1, \mathcal{E}) \supset \mathcal{F}(s_2, \mathcal{E})$. For example, $t_3$ = "Indiana Jones series" is a hypernym of $t_1$, since $t_1$ only maps to a subset of the entities covered by $t_3$.*

**Definition 3 (Hyponym).** *A string $s_1 \in \mathcal{S}$ is a **hyponym** of another string $s_2 \in \mathcal{S}$ over the set of entities $\mathcal{E}$ if and only if $\mathcal{F}(s_1, \mathcal{E}) \subset \mathcal{F}(s_2, \mathcal{E})$. For example, $t_1$ is a hyponym of $t_3$.*

### 4.2.2 Synonym Finding Problem

**Synonym Finding Problem.** Formally, our formulation of the synonym finding problem is as follows. As input, we are given a set of entities $\mathcal{E}$ (*e.g.*, movies) and a set of homogeneous strings (*e.g.*, movie names) $U \subseteq \mathcal{S}$. As output, we would like to produce for each string $u \in U$, its set of synonyms $V_u = \{v \in \mathcal{S} \mid \mathcal{F}(u, \mathcal{E}) = \mathcal{F}(v, \mathcal{E})\}$.

In the problem formulation above, we do not assume that $\mathcal{F}$ is a given. This is because, while we assume that such an oracle function exists (if only abstractly), we do not claim it is obtainable in practice. True $\mathcal{F}$ exists only in the collective minds of all users. Hence, the equality $\mathcal{F}(u, \mathcal{E}) = \mathcal{F}(v, \mathcal{E})$ that underlies Definition 1 cannot be determined exactly.

To resolve this, we propose to relax Definition 1, and instead approximate the equality $\mathcal{F}(u, \mathcal{E}) = \mathcal{F}(v, \mathcal{E})$ using real-life data. We identify the following real-life Web based data sets as especially relevant for this approach:

**Search Data** $A$ consists of a set of tuples, where each tuple $a = \langle q, p, r \rangle$ denotes the relevance score $r$ of a Web page url $p$ for the search query $q \in \mathcal{S}$. For simplicity, in this chapter, we assume $r$ is the relevance rank of $p$, with rank 1 being the most relevant. $A$ captures the "relevance" relationship between a query string and a Web page as determined by a search engine.

**Click Data** $L$ is a set of tuples, where each tuple $l = \langle q, p, n \rangle$ denotes the number of times $n \in N^+$ that users click on $p$ after issuing query $q \in \mathcal{S}$ on a search engine. $L$ captures the "relevance" relationship between a query string and a Web page as determined by search engine users.

How these data sets may be used to find synonyms can be summarized as follows. Let $\mathcal{P}$ be the union of all Web pages, and $\mathcal{Q}$ be the union of all query strings, in $A$ and $L$. Since both $A$ and $L$ represent some form of relationship between query strings and Web pages, we can learn from $A$ and $L$ respectively, two functions $\mathcal{G}_A(q, \mathcal{P}) \rightarrow P$ and $\mathcal{G}_L(q, \mathcal{P}) \rightarrow P$, which map a query string $q \in \mathcal{Q}$ to the subset of relevant Web pages $P \subseteq \mathcal{P}$. Assuming that for any entity $e \in \mathcal{E}$, there always exist Web pages that are appropriate and representative surrogates of $e$ (which is a reasonable assumption given the scope of the Web), we consider it probable that two query strings $q_1$ and $q_2$ are synonyms if $\mathcal{G}_A(q_1, \mathcal{P}) \approx \mathcal{G}_L(q_2, \mathcal{P})$.

**Definition 4 (Web Synonym).** *A string $s_1 \in \mathcal{S}$ is a* ***Web synonym*** *of another string $s_2 \in \mathcal{S}$ over the set of Web pages $\mathcal{P}$ (keeping in mind the actual reference set of entities $\mathcal{E}$) if $\mathcal{G}_A(s_1, \mathcal{P}) \approx \mathcal{G}_L(s_2, \mathcal{P})$.*

**Web Synonym Finding Problem.** In this chapter, our specific problem formulation is as follows. As input, we are given a set of homogeneous strings $U$; the data sets $A$ and $L$; and the reference set of entities $\mathcal{E}$. As output, we would like to produce for each string $u \in U$, its set of Web synonyms $W_u = \{w \in \mathcal{S} \mid \mathcal{G}_A(u, \mathcal{P}) \approx \mathcal{G}_L(w, \mathcal{P})\}$.

## 4.3 A Bottom-Up Solution

To solve the Web synonym finding problem, we propose a two-phase solution, consisting of candidate generation and candidate selection.

### 4.3.1 Candidate Generation

To quickly zoom into synonym candidates for a given $u$, we propose to generate candidate in two steps. First, we seek the Web pages that are good representation for entities referenced by $u$. We term these Web pages *surrogates* of $u$. Second, we find out how users refer to these surrogates.

**Finding Surrogates.** The Web has inarguably become the largest open platform for serving various kinds of data. It is almost certain that entities we have in our entity set $\mathcal{E}$ would have some representation on the Web. This representation (or surrogates) come mostly in the form of Web pages. For a particular digital camera (*e.g.*, Canon EOS 350D), its surrogates may include a page in the manufacturer's site listing its specifications, an eBay page selling it, a Wikipedia page describing it, a page on a review site critiquing it, *etc.*.

Moreover, data appears in various forms on the Web. For instance, a seller on eBay may explicitly list some of the alternative ways to access the data to help increase the chances of her item being retrieved, *e.g.*, "Digital REBEL XT" and "350D". Data, once appearing on the Web, gets enriched in various ways, which enables alternative paths for people to access the same information.

We use the *Search Data* $A$ to find Web page surrogates for a given $u$. $A$ is derived by issuing each $u \in U$ as a query to the Bing Search API and keeping the top-$k$ results. Based on $A$, we can define the mapping function $\mathcal{G}_A(u, \mathcal{P})$ between $u$ to the set of top-$k$ pages, as Eq 4.1 shows.

$$\mathcal{G}_A(u, \mathcal{P}) = \{a.p \mid a \in A,\ a.q = u \wedge a.r \leq k\} \tag{4.1}$$

**Definition 5 (Surrogate).** *A Web page $p \in \mathcal{P}$ is a* **surrogate** *for $u$ if $p \in \mathcal{G}_A(u, \mathcal{P})$.*

It may also be possible to use *Click Data* in place of *Search Data*, whereby a Web page is a surrogate if it has attracted many clicks when the entity's data value is used as a query. However, clicks are not always available for this purpose, as the entities' data values usually come in the canonical form (*e.g.*, the full title name of a movie), and therefore may not be used as queries by people.

**Referencing Surrogates.** Having identified $u$'s surrogates, we next find out how users access those surrogates. Remember that these surrogates are Web pages available for access by the general public. Again, search engine is the primary channel people use for accessing information on the Web. We can therefore regard the queries issued to get to these surrogate pages as the various ways users refer to the entities represented by these pages. Consequently such queries are good synonym candidates for $u$.

Click data $L$ offers us the mapping from candidates to surrogates. Based on $L$, we can define the mapping function $\mathcal{G}_L(w', \mathcal{P})$ between a potential synonym candidate $w'$ to the set of clicked pages, as shown in Eq 4.2.

$$\mathcal{G}_L(w', \mathcal{P}) = \{l.p \mid l \in L,\ l.q = w' \ \wedge\ l.n \geq 1\} \tag{4.2}$$

**Definition 6 (Web Synonym Candidate).** *A string $w'$ is a* **synonym candidate** *for $u$ if and only if* $\mathcal{G}_A(u, \mathcal{P}) \ \cap \ \mathcal{G}_L(w', \mathcal{P}) \neq \emptyset$.

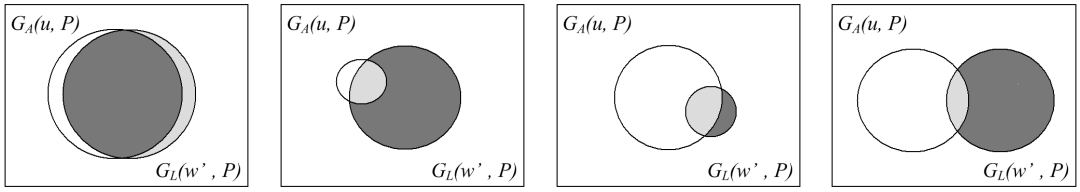
Based on Definition 6, we regard $w'$ as a Web synonym candidate for $u$ if at least one surrogate of $u$ has been clicked when $w'$ is issued as a query. Therefore, the candidate set for $u$ is $W'_u = \{w' \mid \mathcal{G}_A(u, \mathcal{P}) \ \cap \ \mathcal{G}_L(w', \mathcal{P}) \neq \emptyset\}$.

### 4.3.2 Candidate Selection

To estimate the likelihood that a candidate $w'$ is a Web synonym of the input value $u$, we identify two important measures that can be captured from search data $A$ and click data $L$. The two measures respectively capture the *strength* and *exclusiveness* of the relationship between a candidate $w'$ and the input value u.

**Intersecting Page Count (IPC)**. Here, we seek to measure the strength of relatedness between an input value $u$ and a candidate $w'$. In the candidate generation phase, we look for $u$'s candidates by looking at queries ($w'$) for which the following holds: $\mathcal{G}_A(u, \mathcal{P}) \cap \mathcal{G}_L(w', \mathcal{P}) \neq \emptyset$. In Eq 4.3, we derive the $IPC(w', u)$ as the size of this intersection. Intuitively the higher the $IPC$, the larger the size of the intersection is, the more common pages have been referred to using $u$ and $w'$, and therefore the more likely $u$ and $w'$ would be related to one another.

$$IPC(w', u) = |\mathcal{G}_L(w', \mathcal{P}) \cap \mathcal{G}_A(u, \mathcal{P})| \tag{4.3}$$




(a) Synonym (b) Hypernym (c) Hyponym (d) Not Equivalent

Figure 4.1: Venn Diagram Illustration

**Intersecting Click Ratio (ICR)**. Another indicator for the strong relationship between $w'$ and $u$ is if a majority of the clicks resulting from $w'$ as a query land on $u$'s surrogate pages more often than on non-surrogate pages. The click ratio measure $ICR(w', u)$ is determined as shown in Eq 4.4. The higher is $ICR(w', u)$, the more exclusive is the relationship between $w'$ and $u$, and the more likely $w'$ would be a Web synonym of $u$.

$$ICR(w', u) = \frac{\sum_{l \in L,\ l.p \in \mathcal{G}_L(w', \mathcal{P}) \cap \mathcal{G}_A(u, \mathcal{P})} l.n}{\sum_{l \in L,\ l.p \in \mathcal{G}_L(w', \mathcal{P})} l.n} \tag{4.4}$$

We use a Venn diagram illustration in Figure 4.1 to describe how the above two measures work in selecting the best Web synonyms. Consider the example where the input value $u$ is the movie title "Indiana Jones and Kingdom of the Crystal Skull". Figure 4.1(a) illustrates the case where a candidate $w'$ (*e.g.*, "Indiana Jones 4") is a likely Web synonym of $u$. The sets denote the Web pages that are retrieved by $u$ ($\mathcal{G}_A(u, \mathcal{P})$) and are clicked on for query $w'$ ($\mathcal{G}_L(w', \mathcal{P})$) respectively. In this case, the size of the intersection of the two sets is large, indicating a high $IPC$ value. For the set $\mathcal{G}_L(w', \mathcal{P})$, the darkly shaded (resp. lightly shaded) area indicates the subset of pages getting the most clicks (resp. fewer clicks). In this case, most of the clicks fall within the intersection, as opposed to outside of the intersection, indicating a high $ICR$ value. Thus, $w'$ is likely a Web synonym of $u$.

Both $IPC$ and $ICR$ also help to weed out candidates that are related, but not synonyms. Figure 4.1(b) illustrates the case of a hypernym (*e.g.*, "Indiana Jones"). Since a hypernym considers a broader concept, it may be used to refer to many more pages (*e.g.*, concerning other Indiana Jones movies), and consequently most of the clicks fall outside of the intersection (low $ICR$). A hyponym concerns a narrower concept, where there might be more specific pages about the concept outside of the intersection that receive the most clicks (Figure 4.1(c)). Finally, a candidate such as "Harrison Ford" is only related, with low $IPC$ and $ICR$ (Figure 4.1(d)).

We produce the final Web synonym by applying threshold values $\beta$ and $\gamma$ on $IPC$ and $ICR$ respectively.

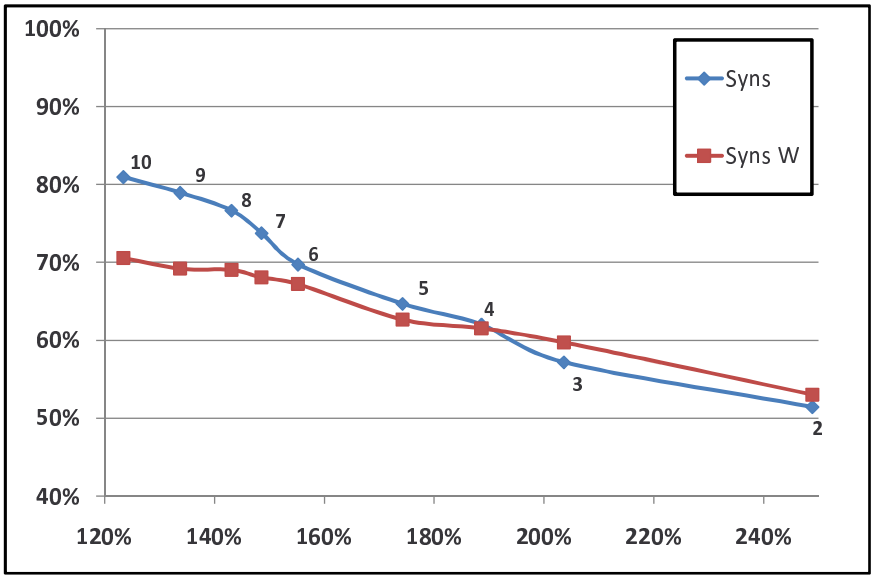



Figure 4.2: IPC Precision and Coverage Increase

## 4.4 Experiments

Our data sets are: D1) the titles of the top 100 movies of 2008 Box office and D2) a collection of 882 canonical camera names crawled from MSN Shopping [57]. All experiments were done on a single windows 2003 server workstation with 8GB RAM and 2TB disk space. We used query and click logs from Bing Search (July to November 2008).

### 4.4.1 Parameter Sensitivity

In this section we evaluate the effect of **Intersecting Page Count (IPC)** and **Intersecting Click Ratio (ICR)** thresholds on *Precision*, *Weighted Precision* and *Coverage Increase*:

| | |
|---|---|
| *Precision* | # of true synonyms over all synonyms generated |
| *Weighted Precision* | Weighted by synonym frequency in query log |
| *Coverage Increase* | Percentage increase in coverage of queries |

We use a precision/recall style figure to show the precision and coverage increase at different thresholds, with $x$ axis for coverage increase, and $y$ axis for precision.

Result for (D1) movies dataset is shown in Figure 4.2, where IPC threshold $\beta$ decreases from left to right on the curves from 10 to 2. We see the higher the IPC, the higher the synonym (Syns) precision is. This effect is a bit weaker in weighted precision. Coverage increase reduces as we increase IPC. Yet, even at high IPC value 10, coverage increase is at 120%, more than doubling the original coverage.

To test the combination of IPC and ICR, we used the threshold values of $\beta \in \{2, 4, 6\}$ for IPC, with ICR

$\gamma$ decreasing from left (0.9) to right (0.01) on the curves, as shown in Figure 4.3 on (D1) dataset. To simplify the figures, we focus on the weighted precision. We see when we increase the ICR parameter, the synonym precision goes up (Syns W 2,4,6). This figure also suggests interesting IPC values around 4, and ICR values (0.1, 0.4, 0.7) acting as local maxima with good balance between precision and coverage increase.

### 4.4.2 Other Approaches to Synonyms

To measure our solution *Us* (thresholds IPC 4, ICR 0.1) against other approaches, we looked at Wikipedia and random walk on the click graph to produce synonyms. We focus on *Hit Ratio* and *Expansion Ratio*:

| | |
|---|---|
| *Hit Ratio* | Percentage of entries producing at least 1 synonym |
| *Expansion Ratio* | Sum of synonyms and orig entries over orig entries |

**Wikipedia**. We use redirection and disambiguation pages in Wikipedia for producing synonyms (*e.g.*, the entry for 'LOTR' redirects to 'Lord of The Rings'). As shown in Table 4.1, Wikipedia performs poorly for less popular entries (*e.g.*, cameras). Our approach consistently creates more synonyms (expansion) and for more entries (hit) for both datasets.

| | Orig | Hits | Ratio | Synonyms | Expansion |
|---|---|---|---|---|---|
| Movies Us | 100 | 99 | 99% | 437 | 537% |
| Movies Wiki | 100 | 96 | 96% | 270 | 370% |
| Movies Walk(0.8) | 100 | 100 | 100% | 229 | 329% |
| Cameras Us | 882 | 767 | 87% | 4286 | 586% |
| Cameras Wiki | 882 | 101 | 11.5% | 576 | 165% |
| Cameras Walk(0.8) | 882 | 479 | 54% | 697 | 179% |

Table 4.1: Hits and Expansion

**Random Walk on a Click Graph**. We used the random walk solution in [36] to evaluate the potential of generating synonyms with default parameters. We see in Table 4.1 that the random walk has low hit ratio on cameras, since the random walk operates completely on the click graph. So if a query has not been asked then no synonym will be produced.

## 4.5 Related Work

WordNet's [58] *synset* (synonym set) feature provides synonyms for a given word or phrase. WordNet synonyms are fundamentally different from the kind of Web synonyms that we are interested in. WordNet synonyms are for common English words, such as those that would be found in dictionaries and thesauri. Thus WordNet will not be able to provide synonyms for movie titles, digital camera names, *etc.*. However,

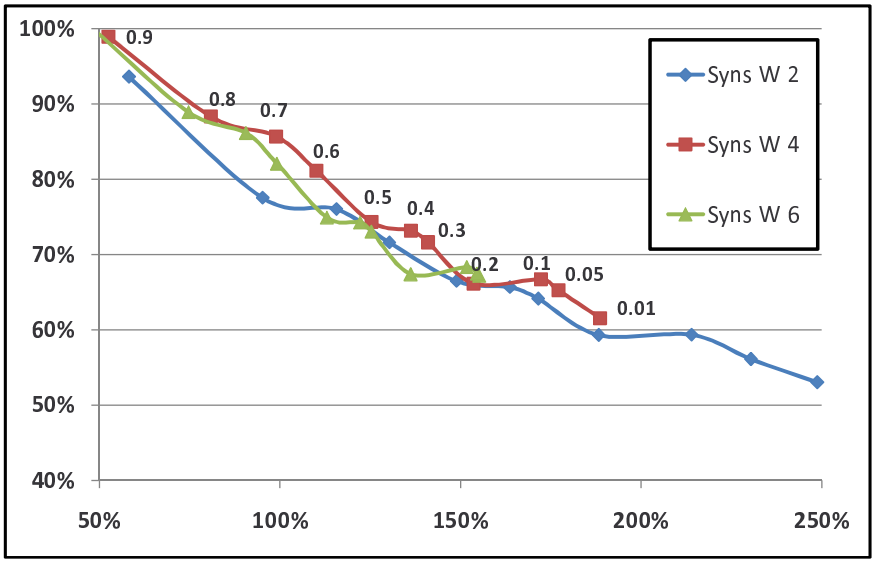



Figure 4.3: ICR Precision and Coverage Increase for IPC 2,4,6

WordNet synonyms can still be useful for enhancing recall in information retrieval [50, 66] when properly applied on the query words that are common English words.

Wikipedia is another online resource that has been proposed to help measure the semantic similarity between a pair of words or phrases. One way to do that is by looking at whether the pair of queries tend to retrieve pages that fall under the same Wikipedia categories [61]. Another way is to exploit the redirection relationship between article titles [40]. Unlike WordNet, Wikipedia does cover some entities (movies, musicians, *etc.*). However, Wikipedia is much smaller than the Web, and therefore the former's coverage of entities are severely limited to only a very small number of the most popular ones, as has been shown in Section 4.4. This limitation motivates us to leverage on the massive data available from the Web and query logs.

[20] is a recent work on finding synonyms by leveraging Web search. Their work focuses on a specific kind of synonym, the ones that are substrings of given entity names. Our focus is more generic in generating many different classes of entity synonyms. Our work relies more on the Web user aspect, in exploiting the alternative ways a user may want to refer to an entity. Another work [53] addresses a related problem, which is to turn a query (*e.g.*, "lord rings") into a displayable form (*e.g.*, "Lord of the Rings") by word reordering, modifier addition, and capitalization. In a way, we can see it as a complementary problem, going in the reverse direction from our work (canonical forms into synonyms), but limited only to those synonyms with significant text similarity to canonical form.

Our work is related to reference reconciliation or record matching techniques ([29, 32, 10, 9]), which

try to resolve different references or records in a dataset to the same real world entity. They are closely related with our work, since we are trying to find the difference references in the form of entity synonyms for a set of given entities. However, there are a few differences from our work. First, they normally assume the "references" are given, while we have to generate the candidate "references" ourselves. Second, such approaches usually rely on multiple attributes to be present to produce high quality results(*e.g.*, name, age, gender for person record). Yet, Web queries normally lack multi-attribute semantic context.

Entity recognition or entity extraction often refers to the problem of identifying mentions of specific entity types (*e.g.*person, location, *etc.*) from unstructured data. [30] and [65] provides a comprehensive overview covering the various information extraction techniques and systems respectively. However, they are different from the problem we are solving in terms of both input and output. Our input is a list of specific entity values. Our output are the alternative names (what we call Web synonyms) for each input value. Entity recognition normally takes input a description of entity type (*e.g.*, a dictionary, patterns or a language model) and aiming at identifying the mentions of the entity type from text.

There are previous works to measure similarity between queries [68] by using Web data, for various purposes such as document ranking [28], semantic relation discovery [7], keyword generation for advertisement [36] and query suggestion ([42, 5]). These similarity based approaches do not work well for our problem for several reasons. First, they may discover many pairs of related queries that are not synonyms (*e.g.*, "Windows Vista" and "PC"). Second, the input for which we seek to derive synonyms are generally well-formed strings as full movie titles or digital camera names, which real users seldom use and may not appear frequently as queries.

Another application for query similarity is query suggestion, which attempts to provide a search engine user with an alternative to the user's current query. For example, for the ambiguous query "jaguar", the search engine may make query suggestions such as "Jaguar Animal", "Jaguar Cars", "Jaguar Cat", etc. There are several ways to derive these suggestions. [42] is based on typical reformulations or substitutions that users make to their queries. [5] further considers whether the query substitutions lead to user clicks on the same ads, while [56] considers whether they lead to clicks on the same webpages. [6] also looks at the similarity of the clicked pages' text content. As previously explained, the notion of similarity used by query suggestion covers a much broader set of relations (not just synonyms), and thus using these techniques to generate synonyms will lead to many false positives.

# Chapter 5

# Related Work

Given we have reviewed related work for each individual tasks in their corresponding chapters, this chapter mainly aims at giving an overview of related systems and contrasting to our effort towards entity-aware search. Our entity search system has a unique focus to enable flexible keyword querying over entities by leveraging the redundancy of the Web, requiring only the extraction of entities.

Towards enriching keyword query with more semantics, AVATAR [43] semantic search tries to interpret keyword queries for the intended entities and utilize such entities in finding documents. Our proposal directly searches over entities and returns promising entities as output.

Towards searching over fully extracted entities and relationships from the Web, ExDB [15, 14] supports expressive SQL-like query language over an extracted database of singular objects and binary predicates, of the Web; Libra [59] studies the problem of searching web objects as records with attributes. Due to the different focus on information granularity, its language retrieval model is very different from ours; NAGA [45] builds a semantic graph based on relationships extracted from webpages, proposes a graph-based query language and studies the underlying ranking problem; Ming [44] further performs mining over the knowledge graph to find out informative subgraphs. While these approaches rely on effective entity and relationship extraction for populating an extraction database, our approach only assumes entity level extraction and replies on large-scale analysis in the ranking process. All the contexts of entities are maintained, which is key to enabling flexible keyword querying.

Towards searching over typed entities in or related with text documents, BE [13] develops a search engine based on linguistic phrase patterns and utilizes a special index for efficient processing. It lacks overall system support for general entity search with a principled ranking model. Its special index, "neighborhood index", and query language, "interleaved phrase" query, are limited to phrase queries only; Chakrabarti et al. [19] introduce a class of text proximity queries and study scoring function and index structure optimization for such queries. Its scoring function primarily uses local proximity information, whereas we investigate effective global aggregation and validation methods, which we believe are indispensable for robust and effective ranking in addition to local analysis. Our query primitive is also more flexible in allowing expressive patterns

and multiple entities in one query; ObjectFinder [17] views an "object" as the collection of documents that are related with it and therefore scores an "object" by performing aggregation over document scores. In contrast, our approach views an entity tuple as all its occurrences over the collection. Therefore, its score aggregates over all its occurrences, where we consider uncertain, contextual factors other than the document score. Bautin et al. [8] propose the idea of concordance document for enabling retrieval of entities. Each entity is represented as a concordance document, which is comprised of all the sentences where the entity is mentioned. This approach could lead to significant blow up of index size, and present difficulty in proximity matching and aggregation which are important for effective entity retrieval.

Our system on one hand relies on information extraction (IE) techniques to extract entities; on the other hand, our system could be regarded as online relation extraction based on association. There have been many comprehensive IE overviews recently ( [27], [30], [4]) summarizing the state of the art. On the special Web domain, there have been many excellent IE system (*e.g.*, SemTag [34], KnowItAll [35], AVATAR [41], SOFIE [62], NELL [16]) Furthermore, many open source frameworks that support IE (*e.g.*, GATE [1], UIMA [2]) are readily available. While most IE techniques extract information from single documents, our system discovers the meaningful association of entities holistically over the whole collection.

Many question answering (QA) systems have resorted to the Web for generating answers. While many QA systems' focus is on developing interesting QA system framework, most of them have adopted simple measures for ranking and lack a principled conceptual model and a systematic study of the underlying ranking problem. The SMART IR system [3] and the AskMSR QA system [11] mainly use the entity occurrence frequency for ranking. The Mulder system [48] ranks answer candidates mainly according to their closeness to keywords, strengthened by clustering similar candidates for voting. The Aranea system [49] mainly uses the frequency of answer candidates weighted by idf of keywords in the candidates as the scoring function. Unlike most QA works (*e.g.*, SMART IR system [3], AskMSR QA system [11], Mulder system [48], Aranea system [49]), which retrieve relevant documents first and then extract answers, our work bypasses the need of retrieving documents and directly builds the concept of entity into the search framework and is therefore more efficient and flexible. Our system can be used as a core component to support QA more directly and efficiently.

# Chapter 6

# Conclusion

The richness of the Web with virtually all kinds of entities, is much beyond the traditional perception of it being a collection of pages. We firmly believe it is an exciting and challenging research area to make the search engines aware of data entities inside pages, and leverage such information to improve users' search experience. Inspired by the success and popularity of keyword based search over documents, my thesis proposes and studies the interesting problem of entity search, by enabling search over entities using flexible keywords.

To begin with, we first deal with the core challenge of effective ranking. To make entity search efficient and scalable, we tackle the challenge of index design and query processing over entities. We further study a related problem of entity synonym discovery to help us discover more entity instances, an important step for supporting the recognition of entities on the Web.

This thesis aims at tackling these challenges towards the goal of enabling entity-aware search. We now summarize the main insights that enable us to conquer these challenges.

First, for effective retrieval of entities, this thesis distills its underlying conceptual model *Impression Model* and develops a probabilistic ranking framework *EntityRank*. Our main insight is the seamlessly integration of both local and global information in the overall ranking process. We systematically distill a set of key characteristics of entity search, *Contextual, Holistic, Uncertain, Associative, Discriminative*, which distinguish it from the traditional document search. Our conceptual model thus captures these key characteristics. Our ranking algorithm concretizes these aspects into computation. Experiments show that capturing these various characteristics is key to the high effectiveness of entity retrieval results.

Second, to meet the requirement of online querying over entities, this thesis studies how to enable efficient and scalable entity search. Our key insight is to treat entity as first class citizen in both indexing design, and query processing. In indexing design, we are mainly inspired by the concept of inverted index. We study the problem from the dual views of reasoning: *entity-as-keyword* and *entity-as-document*. With entity as the first class citizen, we thus consider the inversion from entity to document, as well as the inversion from keyword to entity as indexing principle. This dual indexing principle leads to two index structures, both with

their pros and cons. To achieve high scalability, we exploit the dimension of entity space data partitioning, in addition to the traditional document space data partitioning, for distributing indexes. Naturally, entity search query processing takes advantage of the indexing structures and partition schemes to achieve the requirement of online querying. Our study shows that a synergistic combination of the two index structures and partition schemes lead to nice balance between query efficiency and index space overhead.

Third, to recognize the various instances of entities, this thesis studies the problem of entity synonym discovery. The key insight here is to leverage the wisdom of crowds (both content creators, and end users) on the Web. We notice that content creators, while creating webpages of entities, often try to explicitly spell out the synonyms of entities in order to attract more traffic. At the same time, end users often type in various entity synonyms in order to search for a specific entity and then naturally land on such webpages. We therefore propose to first identify webpages about an entity, and mine over the queries clicked on these webpages for identifying entity synonyms.

In summary, this thesis studied a set of core problems for enabling entity-aware search, proposed novel techniques and systematically evaluated the proposed approaches on large-scale datasets. Through this thesis, we show that enabling entity-aware search is not only an interesting but also a promising direction to enrich the search experience of end users.

# References


[1] Gate - general architecture for text engineering,.

[2] Uima - unstructured information management architecture.

[3] S. Abney, M. Collins, and A. Singhal. Answer extraction. In *ANLP*, 2000.

[4] E. Agichtein and S. Sarawagi. Scalable information extraction and integration (tutorial). In *SIGKDD*, 2006.

[5] I. Antonellis, H. Garcia-Molina, and C. Chang. Simrank++: Query rewriting through link analysis of the click graph. In *VLDB*, 2008.

[6] R. Baeza-Yates, C. Hurtado, and M. Mendoza. Query recommendation using query logs in search engines. In *EDBT 2004 Workshops*, 2004.

[7] R. Baeza-Yates and A. Tiberi. Extracting semantic relations from query logs. In *SIGKDD*, 2007.

[8] M. Bautin. Concordance-based entity-oriented search. *IEEE/ACM Web Intelligence*, 2007.

[9] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *The VLDB Journal*, 18, 2009.

[10] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Trans. Knowl. Discov. Data*, 1, 2007.

[11] E. Brill, S. Dumais, and M. Banko. An analysis of the askmsr question-answering system. In *EMNLP*, 2002.

[12] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, pages 107–117, 1998.

[13] M. Cafarella and O. Etzioni. A search engine for large-corpus language applications. In *WWW*, 2005.

[14] M. J. Cafarella. Extracting and querying a comprehensive web database. In *CIDR*, 2009.

[15] M. J. Cafarella, C. Re, D. Suciu, and O. Etzioni. Structured querying of web text data: A technical challenge. In *CIDR*, 2007.

[16] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. H. Jr., and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.

[17] K. Chakrabarti, V. Ganti, J. Han, and D. Xin. Ranking objects based on relationships. In *SIGMOD*, 2006.

[18] S. Chakrabarti. Breaking through the syntax barrier: Searching with entities and relations. In *ECML*, pages 9–16, 2004.

[19] S. Chakrabarti, K. Puniyani, and S. Das. Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In *WWW*, pages 717–726, 2006.

[20] S. Chaudhuri, V. Ganti, and D. Xin. Exploiting web search to generate synonyms for entities. In *WWW*, 2009.

[21] T. Cheng and K. C.-C. Chang. Entity search engine: Towards agile best-effort information integration over the web. In *CIDR*, 2007.

[22] T. Cheng and K. C.-C. Chang. Beyond pages: Supporting efficient, scalable entity search with dual-inversion index. In *EDBT*, 2010.

[23] T. Cheng, H. Lauw, and S. Paparizos. Fuzzy matching of web queries to structured data. In *ICDE*, 2010.

[24] T. Cheng, X. Yan, and K. C.-C. Chang. Entityrank: Searching entities directly and holistically. In *VLDB*, 2007.

[25] T. Cheng, X. Yan, and K. C.-C. Chang. Supporting entity search: A large-scale prototype search system. In *SIGMOD*, 2007.

[26] J. Cho and S. Rajagopalan. A fast regular expression indexing engine. In *ICDE*, 2002.

[27] W. Cohen. Information extraction (tutorial). In *SIGKDD*, 2003.

[28] N. Craswell and M. Szummer. Random walks on the click graph. In *SIGIR*, pages 239–246, 2007.

[29] D. Dey, S. Sarkar, and P. De. A distance-based approach to entity reconciliation in heterogeneous databases. *IEEE Trans. on Knowl. and Data Eng.*, 14, 2002.

[30] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing information extraction: state of the art and research directions (tutorial). In *SIGMOD*, 2006.

[31] X. Dong and A. Halevy. Indexing dataspaces. In *SIGMOD*, 2007.

[32] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, 2005.

[33] T. Dunning. Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics*, 19, 1994.

[34] S. D. et al. SemTag and Seeker: Bootstrapping the semantic web via automated semantic annotation. In *WWW*, 2003.

[35] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale information extraction in knowitall. In *WWW*, 2004.

[36] A. Fuxman, P. Tsaparas, K. Achan, and R. Agrawal. Using the wisdom of the crowds for keyword generation. In *WWW*, 2008.

[37] J. Guo, G. Xu, X. Cheng, and H. Li. Named entity recognition in query. In *SIGIR*, 2009.

[38] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD*, pages 16–27, 2003.

[39] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword proximity search in xml trees. *IEEE Transactions on Knowledge and Data Engineering*, 18(4):525–539, 2006.

[40] J. Hu, L. Fang, Y. Cao, H.-J. Zeng, H. Li, Q. Yang, and Z. Chen. Enhancing text clustering by leveraging wikipedia semantics. In *SIGIR*, 2008.

[41] T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. *IEEE Data Eng. Bull.*, 29(1):40–48, 2006.

[42] R. Jones, B. Rey, O. Madani, and W. Greiner. Generating query substitutions. In *WWW*, pages 387–396, 2006.

[43] E. Kandogan, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar semantic search: a database approach to information retrieval. In *SIGMOD*, pages 790–792, 2006.

[44] G. Kasneci, S. Elbassuoni, and G. Weikum. Ming: mining informative entity relationship subgraphs. In *CIKM*, 2009.

[45] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. Naga: Searching and ranking knowledge. In *ICDE*, 2008.

[46] G. Kasneci, F. M. Suchanek, M. Ramanath, and G. Weikum. How naga uncoils: searching with entities and relations. In *WWW*, 2007.

[47] R. Kumar and A. Tomkins. A characterization of online search behavior. *Data Engineering Bulletin*, 32, 2009.

[48] C. C. T. Kwok, O. Etzioni, and D. S. Weld. Scaling question answering to the web. In *WWW*, pages 150–161, 2001.

[49] J. J. Lin and B. Katz. Question answering from the web using knowledge annotation and knowledge mining techniques. In *CIKM*, 2003.

[50] S. Liu, F. Liu, C. Yu, and W. Meng. An effective approach to document retrieval via utilizing wordnet and recognizing phrases. In *SIGIR*, 2004.

[51] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *VLDB*, 2003.

[52] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *WWW*, 2005.

[53] A. Malekian, C.-C. Chang, R. Kumar, and G. Wang. Optimizing query rewrites for keyword-based advertising. In *EC*, 2008.

[54] P. Marius, D. Lin, J. Bigham, A. Lifchits, and A. Jain. Organizing and searching theworldwideweb of facts - step one: the one-million fact extraction challenge. In *AAAI*, 2006.

[55] E. Markatos. On caching search engine query results. In *Computer Communications*, 2000.

[56] Q. Mei, D. Zhou, and K. Church. Query suggestion using hitting time. In *CIKM*, 2008.

[57] Microsoft. MSN Shopping XML data access API. http://shopping.msn.com/xml/v1/getresults.aspx?text=digital+camera.

[58] G. A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[59] Z. Nie, Y. Ma, S. Shi, J.-R. Wen, and W.-Y. Ma. Web object retrieval. In *WWW*, 2007.

[60] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10), 1996.

[61] M. Strube and S. P. Ponzetto. Wikirelate! computing semantic relatedness using wikipedia. 2006.

[62] F. M. Suchanek, M. Sozio, and G. Weikum. Sofie: a self-organizing framework for information extraction. In *WWW*, 2009.

[63] D. Taniar, Y. Jiang, K. H. Liu, and C. H. C. Leung. Aggregate-join query processing in parallel database systems. In *HPC*, 2000.

[64] D. Taniar, R. B.-N. Tan, C. H. C. Leung, and K. H. Liu. Performance analysis of "groupby-after-join" query processing in parallel database systems. *Inf. Comput. Sci.*, 168(1-4), 2004.

[65] V. Uren, P. Cimiano, J. Iria, S. Handschuh, M. Vargas-Vera, E. Motta, and F. Ciravegna. Semantic annotation for knowledge management: Requirements and a survey of the state of the art. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4, 2006.

[66] G. Varelas, E. Voutsakis, P. Raftopoulou, E. G. Petrakis, and E. E. Milios. Semantic similarity methods in wordnet and their application to information retrieval on the web. In *WIDM*, 2005.

[67] G. Weikum. Db&ir: both sides now. In *SIGMOD*, 2007.

[68] J.-R. Wen, J.-Y. Nie, and H.-J. Zhang. Clustering user queries of a search engine. In *WWW*, pages 162–168, 2001.

[69] H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.

[70] M. Wu and A. Marian. Corroborating answers from multiple web sources. In *WebDB*, 2007.

[71] W. P. Yan and P.-A. Larson. Performing group-by before join. In *ICDE*, 1994.

[72] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, 2008.

[73] M. Zhou, T. Cheng, and K. C.-C. Chang. Data-oriented content query system: Searching for data in text on the web. In *WSDM*, 2010.

[74] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

# Author's Biography

Tao Cheng was born in China, on February 7, 1981. He graduated from the Zhejiang University with a Bachelor of Science degree in Computer Science in 2003. After a short stay in University of California at Santa-Barbara, Tao then relocated to Champaign, Illinois, to continue the graduate study in computer science. He got his Master of Science degree in Computer Science from the University of Illinois at Urbana-Champaign in 2007 and his Doctor of Philosophy in Computer Science from the University of Illinois at Urbana-Champaign in 2010. Following the completion of his Ph.D., Tao will begin his role as a researcher in Microsoft Research in Redmond.