Compressed Data Structures for Annotated Web Search

Soumen Chakrabarti† Sasidhar Kasturi† Bharath Balakrishnan†
Ganesh Ramakrishnan† Rohit Saraf†

ABSTRACT

Entity relationship search at Web scale depends on adding dozens of entity annotations to each of billions of crawled pages and indexing the annotations at rates comparable to regular text indexing. Even small entity search benchmarks from TREC and INEX suggest that the entity catalog support thousands of entity types and tens to hundreds of millions of entities. The above targets raise many challenges, major ones being the design of highly compressed data structures in RAM for spotting and disambiguating entity mentions, and highly compressed disk-based annotation indices. These data structures cannot be readily built upon standard inverted indices. Here we present a Web scale entity annotator and annotation index. Using a new workload-sensitive compressed multilevel map, we fit statistical disambiguation models for millions of entities within 1.15GB of RAM, and spend about 0.6 core-milliseconds per disambiguation. In contrast, DBPedia Spotlight spends 158 milliseconds, Wikipedia Miner spends 21 milliseconds, and Zemanta spends 9.5 milliseconds. Our annotation indices use ideas from vertical databases to reduce storage by 30%. On 40×8 cores with 40×3 disk spindles, we can annotate and index, in about a day, a billion Web pages with two million entities and 200,000 types from Wikipedia. Index decompression and scan speed are comparable to MG4J.

Category: H.3.3 Information search and retrieval; search process. **General terms:** Algorithms, experimentation, performance. **Keywords:** entity, annotation, indexing, search.

1. INTRODUCTION

Web search today is greatly enriched by recognizing and exploiting named entities, their attributes, and relations between them [6, 10, 19, 21]. Mentions of the entities are embedded in unstructured, "organic" Web pages. In a typical system architecture [13, 24, 12, 26, 20], a *spotter* first identifies short token segments or "spots" as potential mentions of entities from its *catalog*. For our purposes, a catalog consists of a directed acyclic graph of *type* nodes, to which *entity* nodes are attached. Many entities may qualify for a given text segment, e.g., "Michael Jordan" or "Apple". In the second stage, a *disambiguator* chooses among the candidate entities. Collectively, these two stages comprise an *annotator*. An *annotation* record consists of a document ID, a token span, and one or more entities e chosen by the disambiguator, usually accompanied by some score of confidence that the token span mentions that specific e. To assist in search, these annotations need to be indexed. Annotation indices are generally different from standard text indices. Used along with text indices, annotation indices help answer powerful queries that relate types of entities, specific entity literals, words and phrases [20]. E.g., we can collect short windows that include the phrase "p played" within two tokens of m, where p is a physicist and m is a musical instrument. We can then subject these windows to further statistical analysis to generate a ranked list of $\langle p, m\rangle$ tuples.

1.1 The scaling challenge

Text indexing systems have greatly scaled up since the early days of information retrieval. Even public-domain indexing systems like Lucene or MG4J [5] can tokenize and index hundreds of documents per second per CPU core. Such speed is critical to process Web crawls with billions of pages, but it is challenging to preserve it in the face of the substantial extra work involved when indexing the annotated Web: the act of annotation itself, and indexing not just tokens, but also annotated entities and their types. It is critical for the annotation process and the indexing of annotations to be very fast, comparable to text indexing or even faster. While the text in each document version is indexed just once, annotation indexing may be a continual process. As entity catalogs are augmented and cleaned up, and as annotators are actively trained, annotation indexing will usually be performed repeatedly.

SemTag [13], among the earliest annotation systems, processed about 264 million Web pages. However, it used the TAP [15] entity catalog with only 72,000 entities and annotated only two mentions per page on average. In contrast, recent Web annotation systems [24, 12, 26, 20] often use Wikipedia, DBPedia or YAGO [29] with over 200,000 types and two million entities. Our goal is to scale to Freebase (25 million entities) and beyond. Although the corpus is streamed from disk, as the entity catalog scales up, its associated disambiguation model parameters quickly fill up RAM, leaving little space for preparing index runs. Compact data structures, usable by large classes of annotators, are therefore of broad interest.

The Entity Engine [21] and the Dual Inverted (DI) index [9] focus on fast query times by investing more index space. They support only 10–20 types, and do not address compressing data structures for the annotation step itself. We justify supporting large catalogs in Section 2. Yet other research systems [24, 12, 26, 20, 17] focus on quality and not speed or scalability. We shall also see that most public domain offerings of annotation software or services [25, 23] are much slower than the system we present here. We set up terminology and review related work in Section 3.

1.2 Our contributions

In this paper we explore new challenges raised by annotated Web search involving large entity and type catalogs. In Section 2 we use experimental evidence from TREC and INEX to justify the need for large, fine grained type catalogs. This contrasts with earlier approaches [21, 9] that critically relied on just 10–20 very broad types (person, place, phone number, etc.). Toward this end, we introduce and investigate two data structure problems:

- For a large class [24, 12, 26, 20] of statistical annotators, we propose in Section 4 aggressively compressed and yet high-performance, in-memory multilevel maps. To our knowledge, no prior work on statistical disam-

†IIT Bombay; contact soumen@cse.iitb.ac.in

WWW 2012, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1229-5/12/04.

biguation, which is becoming increasingly popular [27, 20, 17], focuses on large-scale data structures suitable for storing model parameters.

- In Section 5 we propose compressed indices for annotations. To avoid disk seeks, we must inline information from the entity mention sites ("snippets") into specially designed posting lists; hence they are called snippet-interleaved-posting (SIP) indices.

Aggressively compressed data structures are critical for fast annotation, because they help us save as much RAM as possible for sorting index runs, which reduces index merge time.

Our system is fully implemented using MG4J [5] components and can process a billion-page corpus on 40 hosts in under a day. In Section 6, we report on large-scale experiments with YAGO's 200,000 types, two million entities, and a 500-million page Web corpus. Each key-value entry in our model map has a uncompressed payload of **128 bits** (three integers mapping to a float); this is compressed down to only **19.2** bits/entry. Nevertheless, we can disambiguate at **0.6** core-milliseconds per spot, orders of magnitude faster than DBPedia Spotlight (**158** ms/spot), Wikipedia Miner (**21** ms/spot), and Zemanta (**9.5** ms/spot), three popular annotators. Our annotation index is 20–30% smaller than using Lucene's *payload* hook. Index decompression rates are competitive with MG4J's posting scan.

2. THE NEED FOR LARGE CATALOGS

For the purpose of this paper, a catalog has a directed acyclic graph of types under the subtype-of relationship, with entities attached to type nodes with the instance-of relationship. Although entity(-aware) search systems [6, 10, 3, 21, 9] are becoming common, no consensus is evident on the number or granularity of types to be supported for the new paradigm to be effective. Prototypes range from one type (persons in expert search) to 10–20 broad types (person, place, sports team, phone number, etc.) [21, 9] to WordNet [6]. Unfortunately, logs from commercial Web search engines, even when available (e.g., Live Labs, AOL), provide only raw string queries, with no explicit indication of the pertinent types.

In an initial attempt to remedy this problem, we had five volunteers identify the YAGO-compliant answer types for entity-seeking queries collected from TREC/INEX over 1999–2006. They succeeded for 889 queries, reporting 315 distinct types. The distribution of answer type frequency vs. rank shows a characteristic heavy tail behavior (Figure 1): collectively, rare types cannot be ignored. Given this is true of modest-sized query sets from two competitions, we anticipate the heavy tail behavior to be only more pronounced in Web queries.

The Dual Inversion (DI) system [9] supports 21 types, among the largest type catalog used in Web-scale entity search. To further study the effect of a limited type catalog, we had volunteers try to map YAGO answer types to DI answer types. As Figure 2 summarizes, they often failed, and even otherwise, the broader type resulted in loss of entity ranking accuracy using a competitive entity ranking system [7]. These results strongly motivate supporting a large number of types.

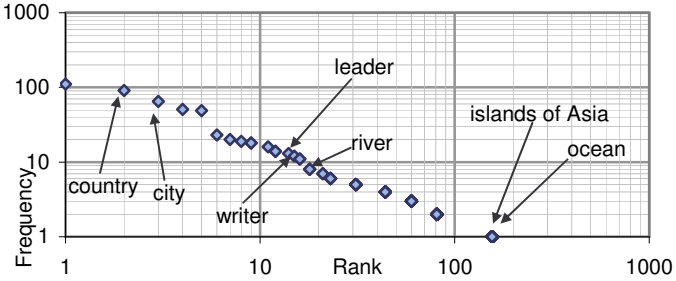


Figure 1: Heavy-tailed type distribution in queries.

Count	Description
889	Queries available with YAGO anwer types
271	Queries where YAGO type had no DI counterpart
190	Queries where NDCG decreased on mapping to DI
52%	Queries with unmappable type or decreased NDCG
0.53	NDCG with YAGO answer types
0.43	NDCG with DI answer types

Figure 2: Adverse effect of mapping from YAGO [29] to coarser [9] answer types.

3. SYSTEM CHALLENGES AND PREVIOUS APPROACHES

Entity annotation to automatically create linked data has become increasingly popular in recent years, leading to several popular annotation software and services, e.g., Alchemy API, DBPedia Spotlight [23], Wikipedia Miner [25], Extractiv, OpenCalais and Zemanta. All of them share the purpose of connecting spans in the text to entity catalogs, though they differ in their algorithms and training data. Most annotators [13, 24, 26, 20, 16, 17], including the ones above, work in two stages: *spotting* (Section 3.2) followed by *disambiguation* (Section 3.3). In our system, we add an annotation indexer (Section 3.4) at the end. In this section we describe the overall framework, the performance challenges, and earlier work.

3.1 Overview

As Figure 3 shows, labeled mention contexts from a *reference corpus* (e.g., Wikipedia) are used to train the disambiguator. Mention contexts from the *payload corpus* (e.g., a Web crawl) are also turned into feature vectors, from which we sample a statistical *workload* that our central data structure, the leaf-feature-entity map ("LFEM") has to process. (The "ℓ, f" notation is explained in Section 3.3.) Because workload data is so sparse, we need to carefully *smooth* the distribution using a test set, which is then submitted to the LFEM compressor (Section 4). Afterward, we scan the whole corpus with the annotator while probing the LFEM. The resulting annotation is sent to an entity and type indexer (Section 5).

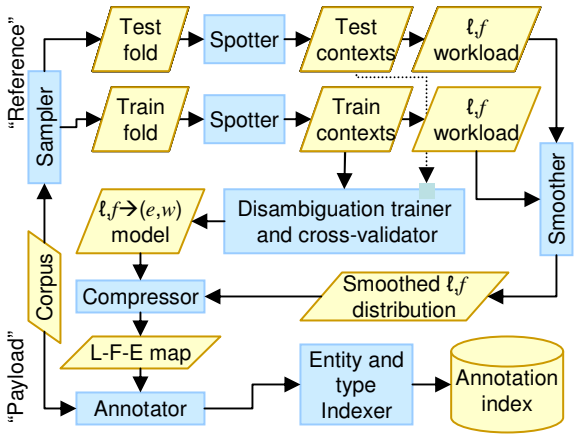


Figure 3: System and workflow overview.

3.2 Spotting

As we scan a sequence of text tokens in a document, the *spotter* flags short token sequences as potential *mentions* of entities in the catalog. Each entity in the catalog is known by one or more (*New York*, *New York City*, *Big Apple*) *canonical phrases* (which may be single tokens). The relation between entities and canonical phrases is many-to-many.

Mentions may match canonical phrases exactly or approximately. Approximate matching [8] is expensive in terms of computation and memory, so Web-scale spotters such as DBPedia Spotlight and others [13, 2] widely use variations on exact match using the Aho-Corasick algorithm or a prefix tree (trie). Spotlight requires **8 GB** to store its matching data structure; this can get problematic trying to scale from DBPedia to Freebase and beyond. Similarly, the Wikify [24] spotter identifies exact phrases to be tagged based on how often they are found in the anchor text of some link internal to Wikipedia. An effective trick [2, Section 3] to ensure good recall in the face of exact matches is to pre-populate the trie with plausible *synonyms* or transformations [25] of canonical phrases. We will assume such an architecture; given this, the time to spot is usually much smaller than the time to disambiguate, which will be our focus.

Each node in the trie represents a partial (prefix) match of a token segment with one or more canonical phrases. Some nodes are designated as *leaves* indicating a completed exact phrase match. Abusing[1] the word "leaf" (because it is a short name good for use in code), trie leaves can have children, in case one dictionary phrase is a prefix of another. E.g., *New York Times* and *New York University* are children of *New York*, and they are all leaves. In general, any spotter will have an analog to a *leaf* ℓ: an artifact that

- expresses a suitable match between a potential mention and a canonical phrase in the catalog, and
- lets us access a set of *candidate entities* E_ℓ that may be mentioned by the canonical phrase corresponding to ℓ.

Therefore, our architecture and notion of a "leaf" are quite generic, and not tied to the use of a trie.

3.3 Disambiguation

After spotting, disambiguation consists of choosing some entity $e \in E_\ell$ as the most likely candidate. Sometimes, the correct decision will be to *reject* the mention, i.e., not link it to any entity in E_ℓ from the catalog (e.g., many John Smiths mentioned in Web text that are not represented in Wikipedia).

The mention is embedded in a *context*, which could range from the neighboring tokens to the entire text on the document, to the site or domain to which the document belongs. For the purpose of disambiguation, the context is distilled into a context *feature vector* $\mathbf{x} \in \mathbb{R}^d$ for a suitable number of features d. Summarizing the notation,

- $\mathbf{x}$ denotes the feature vector distilled from the potential mention.
- ℓ denotes the trie leaf that has been activated by scanning $\mathbf{x}$.
- E_ℓ is the set of entities, including the reject option, that are eligible for linking to the mention.

For each $e \in E_\ell$, we train *weights* $\mathbf{w}_{\ell e} \in \mathbb{R}^d$. The *score* of entity e is the inner product $\mathbf{w}_{\ell e}^\top \mathbf{x}$. The entity chosen among the candidates is $\arg\max_{e \in E_\ell} \mathbf{w}_{\ell e}^\top \mathbf{x}$. This form of inference can model naive Bayes, logistic regression and multiclass SVM classifiers, as well as the local (node potential) part of collective disambiguation techniques [20]. Effectively, disambiguation amounts to a multiclass (with the reject option) labeling problem at each leaf of the trie, and each context is an instance. For simplicity and concreteness we will focus on naive Bayes disambiguators. Preliminary experiments suggest that generative models are not only faster to train but also more accurate than SVMs, given the extreme sparsity of training data.

3.3.1 Feature space

For each ℓ, e, $\mathbf{w}_{\ell e}$ is a map $f \to w$ from features to model weights. We will explain in Section 4 that it is best to organize this map as $\langle \ell, f \rangle \to \{e \to w\}$. We call this the leaf-feature-entity map, or LFEM. In our implementation, we use as features f:

- Words that appear in the immediate lexical neighborhood of the mention, i.e., within some number of tokens to the left and right of the mention. Experiments suggest that these are necessary, but not adequate, for high-accuracy annotation.
- Salient words from the whole document where the mention is embedded. Salient words are those that contribute a large fraction to the norm of the document in TFIDF vector space.

In this setting, the feature space is *twice* the size of the reference vocabulary, because a nearby word is coded differently from the same word appearing far away in the same document. Overall, we use several million features (although each feature vector is very sparse). This is typical of modern information extraction techniques [27].

3.3.2 Weight map in RAM with fast random access

Critical to fast disambiguation is holding the LFE map in RAM, as compactly as possible. This is challenging: even in our modest testbed, there are over a million leaves ℓ, over two million entities e, and several million fs. There are about 480 million (ℓ, f, e) keys. (Note that the 96-bit key does not fit in any primitive type.) Using standard maps from JDK, Gnu Trove, Colt, or MG4J [5] are all out of the question; just the keys fill up ∼6GB of RAM unless we compress them aggressively, and hash table overheads would easily double the space needed. Surprisingly, current literature on entity disambiguation [24, 12, 26, 20] provides little or no information about data structures and performance, which is the focus of this paper.

While achieving good block compression, we also ensure fast random access by inserting *sync points* for decompression, carefully chosen to minimize expected query time. Workload-sensitive linked binary search trees [18] and skip lists [22] are known. However, their access costs are modeled as the expected number of pointers traversed, not scanning and decompression costs. Boldi and Vigna [4] use statistics from inverted lists[2] to insert inlined optimal skip lists in inverted lists. Unlike earlier work, they do focus on compression, but do not consider access workload distributions. Chierichetti *et al.* [11] consider the specific access workload of sequentially merging inverted lists and insert optimal skip pointers for best expected query time. Our sync point allocation problem is different from the above.

[1] Or we may use "leafy nodes" instead of "leaf nodes".

[2] Skip pointers in inverted lists are different from skip lists.

As the catalog becomes arbitrarily large, it is possible that the disambiguation model will overflow RAM. In that case we need to partition the catalog and make multiple passes through the corpus. Even then, it is critical to compress the model aggressively, to minimize the number of passes. Therefore our basic problem remains well-motivated regardless of the size of the catalog.

3.3.3 Other disambiguation systems

Most other systems [13, 24, 12, 26, 17, 23] depend on some computation of similarity between a test context and entity-labeled training contexts, and this can benefit from our work. However, data structure details are sparse. DBPedia Spotlight uses Lucene to index words in contexts of reference mentions. The context of a test mention is used as a query to fetch the most similar reference mentions, thus implementing a form of k-nearest-neighbor classification. Unless (or even if) the whole index is cached in RAM (explicitly via Lucene options or implicitly via OS buffers), this approach is unlikely to beat our highly customized disambiguation data structures. We will confirm this in Section 6.1.6.

3.4 Indexing annotations

Each annotation record contains a document ID, a token span, the leaf (ID) matched by the span, a subset of the candidate entities associated with the leaf, and a confidence score associated with each candidate. Standard inverted indices map from keywords to posting lists that contain document IDs and offsets within documents where the keywords occur. Postings for entities are similar, except that 1. a mention may span more than one token, so the left and right boundaries need to be recorded, and 2. additional fields like the leaf and confidence scores need to be encoded. We also need posting lists keyed by types [6]: if e (e.g., Albert Einstein) is mentioned at a given span in a given document, any type T such that $e \in T$ (e.g., theoretical Physicist) also appears there. This may greatly enlarge the index, but techniques are known to limit the size [6]. The above are all conventional *document-inverted* indices where keys are tokens, entities or types, and postings are sorted primarily by document ID for DAAT query processing.

For systems that support very small type catalogs [21, 9], one can use clever *collocation* (also called *entity-inverted*) indices that are keyed on $\langle \text{token}, \text{type} \rangle$ and record all windows of limited width where the token and the type co-occur, along with the specific entity ID. This trick can significantly speed up some queries, but requires a great deal of extra index storage, and may not help for multi-predicate queries on a cluster. As we shall see in Section 6.2, large type catalogs lead to collocation indices that are 36× larger than a DAAT index. (The proposal [9] is to limit the collocation index to only some $\langle \text{token}, \text{type} \rangle$ pairs, but no specific algorithms are suggested.) Another issue that is not addressed in earlier work [6, 21, 9] is how all the ancillary fields in type indices should be compressed.

4. DISAMBIGUATION DATA STRUCTURES

Recall from Section 3.3 that conceptually, the LFEM is a map from ℓ, f, e to a suitable model weight w, where ℓ is a leaf, e is an entity, and f a feature. In most recent systems, all three would be represented as 32-bit integers, unless compressed. The value, w, may differ with the kind of machine learning algorithm used: in case of naive Bayes classifiers it would be an integer (typically small) and in case of logistic regression or SVMs it would be a floating point number.

4.1 The three-level $\ell, f, e \to w$ map

Most disambiguation algorithms will proceed as follows as they scan document tokens:

```
foreach trie leaf ℓ encountered during token scan do
    use context to build feature vector x = (x_f)
    initialize score accumulators for each e ∈ E_ℓ
    foreach feature f do
        probe LFEM with key (ℓ, f); get {e → w} map
        foreach entity e do
            update score accumulator of e using w, x_f
```

Note that ℓ comes from a dense ID space, because leaves of the trie can be assigned contiguous IDs, whereas, for any given ℓ, the sets of fs and es encountered are sparse. Therefore, it makes sense to store the $\ell, f, e \to w$ map as $\langle \ell, f \rangle \to \{e \to w\}$, i.e., probe using ℓ, f as key and get back a sparse map from e to w.

We will now use ideas from standard inverted index compression and dictionary coding from column-oriented (vertical) databases [31, 1] to design a compressed version of the $\langle \ell, f \rangle \to \{e \to w\}$ map. Working outward from the most detailed level:

- Sort the entity IDs in E_ℓ in some canonical order (say, decreasing reference corpus frequency). In the context of the current ℓ, let the rank of e in E_ℓ be called e', or the *short entity ID* of e. Note that even if es themselves range in the millions, e's are small numbers, typically under five for most leaves.
- For every (ℓ, f) we will store a table of es and ws. es will be sorted in short ID order and each e_2 will be encoded as $\gamma(e_2' - e_1' - 1)$, the gamma code of the gap between e_2 and the previous entity e_1's short IDs, minus one. This means that for consecutive entities we will spend only one bit in recording e_2.
- For a Bayesian disambiguator, w is an integer and will be gamma encoded. For others, compact discretizations are known [28].
- The $\{e' \to w\}$ table varies in size for various (ℓ, f) keys. For a fixed ℓ, we will sort fs in increasing order and write down $\gamma(f_2 - f_1 - 1)$ (but no short IDs this time). After this we will record the number of bits in f_2's entity-weight table.
- Each block corresponding to a leaf is also of irregular size. Because ℓs are from a compact, dense ID space, we can keep an in-memory array from ℓ to the bit offset where ℓ's block ends.

Throughout, we will be referring to offsets as *bit* offsets. This is efficiently supported by `InputBitStream` and `OutputBitStream` in MG4J [5]. Here, and in Section 5, we will use gap/gamma codes as a first prototype implementation. Preliminary experiments suggest that about 10% further savings are possible using a tuned Golomb code; details are deferred to a full version of this paper.

4.2 Optimal sync tables for random access

While the above scheme achieves good compression, decompression involved in responding to an (ℓ, f) probe can be very slow, because we have to

- Locate the beginning of the leaf block for ℓ.

- Scan forward in ℓ's block until we find the sub-block for f.
- Load and return the $\{e \to w\}$ map block.

Although many ℓs will have short blocks, we expect plenty of large leaves (meaning, leaves with large blocks) as well. Worse, assuming there is some coherence between the reference and payload corpora, we expect the frequently encountered leaves to be large.

A standard remedy [30] is to prepare, for a small, carefully-chosen subset of features, a *sync table* with fixed-size integers, telling us where the $\{e \to w\}$ table for feature f begins. Given a feature f, we binary search the sync table to identify a (hopefully) short segment of the leaf block to scan, rather than the whole leaf block. The key issue is:

> Given an upper limit to the size of the sync table, divide it among the leaves, and then, for each leaf, choose the features to be included in the sync table, so as to minimize the expected or total query time during annotation.

In principle, we can solve a single global optimization that allocates a global sync budget over all leaves simultaneously, but with millions of leaves and typically hundreds of features per leaf, this would be computationally impractical. Therefore we structure our allocation as an "outer" distribution of the global budget to leaves (Section 4.6), followed by an "inner" sync allocation within each leaf (Section 4.4). While the trade-off between sync budget and random access cost is standard [30, page 177], we are not aware of a prior two-level formulation of a workload-driven sync optimizer like ours.

4.3 Workload model

The sync allocator (part of the "Compressor" block in Figure 3) must be informed by two kinds of input:

- The number of bits in each $\{e \to w\}$ map, and the total number of bits in the bit block associated with ℓ.
- The relative rate or probability with which each specific (ℓ, f) is probed in the LFEM. We can normalize the probe counts into p_ℓ, the marginal probability of hitting ℓ, and $p_{f|\ell}$, the (conditional) probability of hitting f inside the leaf block for ℓ.

4.3.1 Feature snapping

The LFE map is prepared from a *reference corpus* where mentions are manually tagged to entities. The contexts of these mentions supply the features that appear in the LFE map. Inevitably, training data is small and sparse. When the payload corpus is scanned, many features are encountered that were never registered in the LFE map. In NLP, these would be called "out of vocabulary" (OOV) features. From the way we access the LFE map, it is clear that an OOV feature f probe results in scanning past to the next larger feature, realizing that f is not there in the map. Therefore, in characterizing workload distributions, we will "snap" OOV features in the payload to the next larger feature in the LFE map built from the reference corpus.

4.3.2 Feature distribution smoothing

We must tune the annotator's performance by sampling a small fraction of the corpus. Therefore, missing leaves, or leaves with poorly characterized feature distributions, are normal and expected. Figure 4 shows the fraction of 1,162,488 leaves hit at least once by an increasing number of sample documents from our 500 million document corpus. At 20 million documents, ~75 percent of leaves are hit. The situation with feature hit distributions within leaves is similar, generally even more extreme. To cope, we use a train-test sampler (Figure 3) with Lidstone (also called Dirichlet) smoothing. Fix a leaf with F features, and let n_f ($\hat{n}_f$) be the number of hits on feature f in the training (test) set. Then we model $\Pr(f) = \frac{\lambda + n_f}{F\lambda + \sum_\phi n_\phi}$, for a parameter λ, tuned empirically as

$$\arg\max_{\lambda > 0} \sum_f \hat{n}_f \log \frac{\lambda + n_f}{F\lambda + \sum_\phi n_\phi}. \quad (1)$$

The idea is that if n_fs are very sparse, λ needs to be large so that OOV features in $\{\hat{n}_f\}$ get adequate probability. We observed diverse λ to be suitable for different leaves.

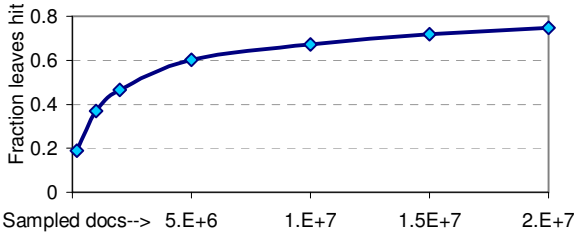


Figure 4: Leaf coverage as number of documents sampled is increased.

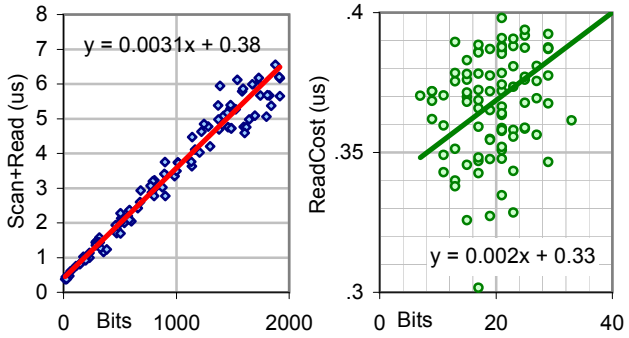


Figure 5: Scatter plots of scan+read and read cost per bit in the $e \to w$ maps ("us"=μsec).

4.4 The inner sync allocation problem

Given a sync budget for each leaf, the inner policy distributes the budget of a leaf among the features in the leaf block. We first need a performance model that predicts the expected cost of a given choice of syncs. To develop this model, we introduce the following notation.

$p(f)$: The probability of querying for feature f.
$b(f)$: The number of bits in the e-to-w map for f.
s_0: The fixed time taken to initiate a scan operation.
s_1: The incremental time taken to scan a bit. To scan β bits, $s_0 + s_1\beta$ time is needed.
r_0: The fixed time to initiate reading an e-to-w map.
r_1: The incremental time taken per bit to read an e-to-w map. The time taken to read the map for f will therefore be $r_0 + r_1 b(f)$.

The linear cost assumptions are reasonably justified by measurements on our system, shown in Figure 5.

4.4.1 Extreme inner policies

Equispaced syncs (Equi). Suppose the training data has *zero* feature hits for a given leaf, i.e., all $n_f = 0$. As is visible from Figure 4, there will always be many such leaves. In such a situation, a reasonable default approach is to place syncs approximately at a regular gap (measured in bits).

Equispaced syncs ensure that no feature is too far from a sync feature, but it does not pay attention to the rate of hitting different leaves and features when scanning the payload corpus. We call this the **Equi** inner policy.

High-frequency syncs (Freq). At the other extreme, if we have so much n_f data at a leaf that we suspect $\hat{n}_f$ will be very similar, we can allocate the leaf's budget entirely to the features having the largest n_fs. We call this the **Freq** inner policy. Our initial intuition was that leaves with enough hit samples are also likely to dominate ℓ, f query times, and so overall Freq should win over Equi. Section 6.1 describes some interesting surprises and retrospective insight.

Simple hybrid policies. We need not fix a single policy for all leaves. We can simply choose the best of Freq and Equi at each leaf. We call this the **FreqOrEqui** inner policy; this gave a very modest gain. Another simple option is to hedge our bets and allocate half (or some tuned fraction of) the budget to frequent features and the other half to equispaced features. We call this policy **EquiAndFreq**.

While the above policies can be implemented very efficiently, they are unsatisfactory in the sense that they are not guided by a precise cost model. We take this up next.

4.4.2 Cost-based optimization

For a fixed leaf, we will fill up a cost table $B[u,k]$ and a backtrace table $L[u,k]$. Here u is (an index to) a feature and k is a budget value. Say u ranges between 0 and $F-1$ and k ranges between 0 and K_ℓ, the budget for this leaf. (Here we drop ℓ because it is fixed.) $B[u,k]$ is the best cost for probes up to and including feature u, using exactly k syncs. We finally want $B[F-1,K]$.

We start by filling in the first column $B[\cdot,0]$ corresponding to $k=0$ (no sync allocated). To retrieve the map for feature i, we

- Scan $\sum_{j=0}^{i-1} b(j)$ bits in time $s_0 + s_1 \sum_{j=0}^{i-1} b(j)$.
- Then read the map in time $r_0 + r_1 b(i)$.

Given zero sync budget, we just have to prefix-sum the above

$$B[u,0] = \sum_{i=0}^{u} p_i \left(r_0 + r_1 b(i) + s_0 + s_1 \sum_{j=0}^{i-1} b(j) \right)$$

By convention, $B[u,\cdot] = 0$ for $u < 0$. Also, in filling $B[u,k]$, budget k *must* be used up, so we will set $B[u,k] = \infty$ if $u < k-1$. To fill $B[u,k]$ for $u \geq k-1$, we can place the rightmost (kth) sync at position ℓ, where $k-1 \leq \ell \leq u$. Then

- The cost of accesses left of the sync at ℓ is $B[\ell-1,k-1]$.
- Accessing the record at ℓ involved no scans[3] and only the read cost of $r_0 + r_1 b(\ell)$.
- For any feature $i = \ell+1, \ldots, u$, the scan cost will be $s_0 + s_1\big(b(\ell) + \cdots + b(i-1)\big)$, followed by a read cost of $r_0 + r_1 b(i)$.

Multiplying with probabilities and summing up, we get

$$B[\ell-1,k-1] + r_0 \textstyle\sum_{i=\ell}^{u} p(i) + r_1 \sum_{i=\ell}^{u} p(i)b(i)$$
$$+ s_0 \textstyle\sum_{i=\ell+1}^{u} p(i) + s_1 \underline{\sum_{i=\ell+1}^{u} p(i)\big(b(\ell) + \cdots + b(i-1)\big)}$$

All but the last sum is trivial to evaluate in $O(1)$ time per cell of B, by computing some prefix sums ahead of time. For convenience we write $P_i^j = p(i) + p(i+1) + \cdots + p(j)$. (If $i > j$, then $P_i^j = 0$.) The last term underlined above is 0 for $\ell = u$. Writing out the last term in detail for $k-1 \leq \ell < u$:

$$\begin{array}{lllll} p(\ell+1)b(\ell) & + & & & \\ p(\ell+2)b(\ell) & + & p(\ell+2)b(\ell+1) & + & \\ \vdots & + & \vdots & + & \ddots \\ p(u)b(\ell) & + & p(u)b(\ell+1) & + \cdots & +p(u)b(u-1) \end{array}$$
$$= P_{\ell+1}^u b(\ell) + P_{\ell+2}^u b(\ell+1) + \cdots + P_u^u b(u-1),$$

it is now seen possible to evaluate it using prefix sums. The pseudocode for filling table B and tracing back the syncs is given in Figure 6; it takes $O(F^2K)$ time, down from $O(F^3K)$ if the above trick were not used. In addition, if we limited candidate syncs to $C \ll F$ positions, we can improve to $O(CFK)$ time (details omitted).

```
Allocate syncs:
prepare prefix sum of p(i) and p(i)b(i)
initialize B[·, 0]
for k = 1, 2, ..., K do
    for u = 0, 1, ..., F − 1 do
        let B* ← ∞ and L* ← ⊥
        if u ≥ k − 1 then
            let B? = B[u − 1, k − 1] + r0 p(u) + r1 p(u)b(u)
            if B* > B? then
                B* ← B? and L* ← ℓ
        initialize g ← 0
        for ℓ = u − 1, ..., k − 1 do
            g ← g + P_{ℓ+1}^u b(ℓ)
            B? ← B[ℓ − 1, k − 1] + s1 g + (r0, r1, s0 terms)
            if B* > B? then
                B* ← B? and L* ← ℓ
        B[u, k] ← B* and L[u, k] ← L*
Backtrace syncs:
k ← K
for ℓ = F − 1, F − 2, ..., 0 do
    if k < 0 then
        break
    ℓ' ← L(ℓ, k)
    if ℓ' ≥ 0 then
        record ℓ' as the kth sync
    ℓ ← ℓ' − 1, k ← k − 1
```

Figure 6: DynProg inner sync allocation.

4.5 Feature ID reordering

When contexts are turned into sparse feature vectors, common practice is to clock up an ID generator to allocate feature IDs to new features, as they are encountered. This policy tends to assign small IDs to frequent features because they are encountered sooner. Since IDs are arbitrary, we can help along further by assigning IDs more deliberately. This is reminiscent of (but different from) optimal document ID assignment for maximally compressed inverted lists [14].

Suppose, within a given leaf, we reordered feature IDs by decreasing order of their occurrence frequency or probability $p(f)$ while annotating the payload corpus. Because the most frequent IDs are close to each other, just a few syncs may suffice to drastically reduce the average amount of scanning and decompression required.

However, recall that the $e \to w$ maps for different feature occupy different number of bits $b(f)$. The features with largest $p(f)$ may also have large $b(f)$, pushing out later fea-

[3]Strictly speaking, we have to binary search the sync table, but this cost is negligible compared to decompressing maps.

tures by many bits, increasing scan costs. This suggests sorting features by something like $p(f)/b(f)$.

The main problem with the above proposals is that we cannot afford one feature permutation per leaf, because we expect to handle tens to hundreds of millions of leaves. Here we settle for one global feature permutation. In Section 6.1.4 we will present some surprising effects of reordering features.

4.6 The outer sync budgeting problem

Suppose K is the global budget over all leaves, in terms of the number of syncs we can afford. The budget is divided among leaves, say leaf ℓ gets K_ℓ syncs. Dividing K into $\{K_\ell\}$ is the job of the *outer policy* and the topic of this section. Once K_ℓ syncs are allocated to ℓ, deciding which features get sync records is the job of the *inner policy*, which was discussed in Section 4.4.

Let b_ℓ be the number of bits in the block for ℓ. Intuitively, we should allocate more syncs to ℓ if either p_ℓ is large or b_ℓ is large, although a large b_ℓ by itself may not motivate many syncs. This suggests the following baseline heuristics:

Hit: $K_\ell \propto p_\ell$

HitBit: $K_\ell \propto p_\ell b_\ell$

Following Witten *et al.* [30, page 177], next we give a more careful outer policy based on these simplifying assumptions:

- $p_{f|\ell}$ is uniform over all f for any ℓ.
- The inner policy inserts syncs at uniform gaps.

Under the above conditions, on an average, a feature probe will involve a scan over $b_\ell/2K_\ell$ bits, starting at a sync. It is reasonable to model the cost of scanning from a sync to the desired feature as proportional to the number of bits scanned. Therefore the overall cost is proportional to $\sum_\ell p_\ell \frac{b_\ell}{K_\ell}$, and we wish to find

$$\arg\min_{\{K_\ell\}} \sum_\ell \frac{p_\ell b_\ell}{K_\ell} \quad \text{s.t.} \quad \sum_\ell K_\ell = K. \tag{2}$$

Using standard Lagrangian optimization, at optimality,

$$\frac{p_\ell b_\ell}{K_\ell^2} = \text{constant} \quad \text{or} \quad K_\ell \propto \sqrt{p_\ell b_\ell} \tag{3}$$

which we call the **SqrtHitBit** policy. Curiously, even though the assumptions made here are not valid, SqrtHitBit performed significantly better than other outer policies. We experimented with other forms of diminishing return curves but these did not result in convincing improvements.

5. COMPRESSED ANNOTATION INDEX

Continuing from Section 3.4, here we describe compact document-inverted indices for annotations. The annotation index has two parts: the *entity* index and *type* index. We will directly proceed to describe type indices because they are a generalization of entity indices. The key is a type from the catalog. The posting list contains a block for every document where an entity of the key type is potentially mentioned. Each document block has to record the token spans of the potential mentions. In a standard term or entity index, the key itself tells us *what* term or entity occurred at an offset. But for a type index, we need to retrieve the specific entity that was mentioned in a span. It is impractical, except in the smallest of corpora [6], to seek and fetch the document, so we need to inline the entity ID within the posting. We call this general style *snippet interleaved postings* (SIP) coding. Here we discuss only the SIP coding for the entity ID; other fields (e.g. annotation confidence) will be discussed in a full version of the paper.

As in Section 4.1, we will be using gap/gamma codes as a first prototype. We are investigating gap distributions and other codes in ongoing work.

5.1 Separate postings by entity (EntSIP)

Suppose three entities *Einstein*, *Feynman* and *Hawking* of type *scientist* are mentioned in a document at various offsets/spans. Note that the IDs assigned to these entities in the catalog are usually arbitrary. E.g., Wikipedia and derivative catalogs have a few million entities, each usually assigned a 32 bit ID (although about 23 bits may suffice). One way to organize the block for this document in the posting list keyed by *scientist* is to write down

- the document ID, gap-gamma coded wrt the previous document
- the number of distinct scientists mentioned in the document in gamma code (because it tends to be a small number)
- write a sub-block for each distinct entity, which contains
  - the entity ID in standard binary code, which we will call the "long" entity ID hereafter
  - the posting list of spans, each expressed as ℓ, r where ℓ, the left endpoint of the span, is gap-gamma coded wrt the previous span's left endpoint, and r is gap-gamma coded wrt ℓ.

Note that delimiting sub-blocks will consume some bits. The advantage of the above code is that each long entity ID is written down exactly once. However, a potential shortcoming is that, by partitioning the posting list of *Scientist* into one list per entity, the gaps in each list become larger.

5.2 Merged postings over all entities (DictSIP)

On the other hand, if we wish to compress the gaps better by writing down a single merged posting list, we need to embed the entity ID better than its "long" version. Inspired by ideas from vertical (column-oriented) databases [31, 1], we employ a per-document dictionary mapping between "long" and "short" entity IDs. In our running example, if *Einstein*, *Feynman* and *Hawking* appear in the document in order of decreasing frequency, we assign them "short" entity IDs of 0, 1, 2 (for the current posting list only; for other lists they may take on different short IDs). Now a document block looks like this:

- The dictionary is simply its size 3 (gamma coded) followed by three long entity IDs.
- The number of posts in this document's posting list.
- For each post ℓ, r, e
  - ℓ coded gap-gamma wrt the previous ℓ
  - r gap-gamma-coded wrt ℓ
  - The short entity ID of e, which is just the position of e in the dictionary. The short ID is gamma coded.

6. EXPERIMENTS

Our corpus is very similar to ClueWeb09, with about 500 million mostly-English pages. Our annotation and indexing code uses MG4J [5] extensively and runs on 40 HP DL160G5 computers, each with 8 2.2GHz Xeon cores, 8GB DDR2 RAM, and four 750GB 7200rpm SATA drives. Most of our code is written to use all cores efficiently. From the 500M corpus, we drew disjoint train+test samples of size 2+2M, 4+4M, 6+6M, 8+8M and 10+10M for cost modeling, but the resulting annotator was deployed on the whole corpus.

6.1 LFEM performance

We begin with a summary of LFEM's RAM requirement with YAGO as catalog. If the map was stored with keys ℓ, f, e and values w, there would be 478,163,567 entries. If ℓ, f, e, w were each assigned 32 bits (int or float), $\mathbf{7.64 \times 10^9}$ bytes would be needed at **128** bits/entry. Based on separate marginal distributions over each of ℓ, f, e, w, the self-information is **33.6** bits/entry. Hash maps from JDK, Gnu Trove or COLT would more than double the RAM needed. (16 GB may not sound large for modern servers, but scaling up from 2 to 25 million entities would then become quite impossible.) In contrast, our LFEM takes only $\mathbf{1.15 \times 10^9}$ bytes, or an average of only **19.2** bits/entry. In other words, each 32-bit number (int or float) was compressed down to 4.8 bits on average.

6.1.1 *Inner policies compared*

Figure 7 compares various inner policies keeping other choices (such as outer policy) at their best settings. The policies compared are Freq, Equi, EquiAndFreq (which is a half-half[4] blend of frequent and equispaced features), and DynProg (which bails out to Equi if the leaf problem is too large to solve in about a minute, which was the case for about 3–4% of the leaves).

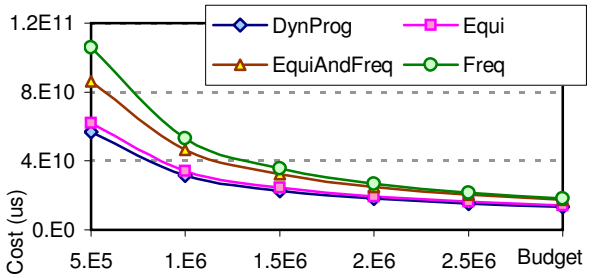


Figure 7: Inner policies compared ("us"=μsec).

The first surprise to us was that Equi was uniformly better than Freq, when not only the sync budget but also all other policies and parameters were varied. We expected total cost to be dominated by large leaves with plenty of training data, where Freq would do well. However, this was not the case. A closer study revealed that, out of 1,162,488 leaves,

- Freq and Equi were tied on 175,001 leaves.
- Freq was cheaper than Equi for 221,341 leaves. In these leaves, the average number of features was 256, and the difference of cost (Equi minus Freq) was 1.93×10^8.
- Equi was cheaper than Freq for 143,564 leaves. In these leaves, the average number of features was 2231, and the difference of cost (Freq minus Equi) was 8.22×10^9.

(The remaining leaves did not occur in the diagnostic sample.) Thus, Freq does better in more leaves, but the gain is minuscule compared to its losses in other leaves, because leaves where Freq wins tend to have many fewer features and lower cost. For large leaves with many features, the heavy tail effect makes Equi's conservative approach win to Freq's "rote learning".

The above analysis suggested that blending Freq and Equi might give a solution better than both of them, but this is also clearly not the case: blending gives costs strictly between Equi and Freq. The optimal dynamic programming allocator beats Equi by a modest margin. It is encouraging to verify that Equi is close to the optimal, because Equi allocation is algorithmically trivial and very fast (millions of leaves per minute).

[4]We also tried other proportions.

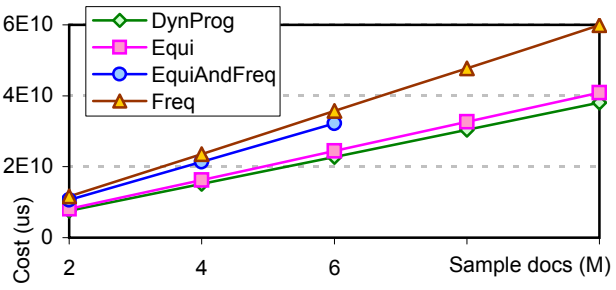


Figure 8: Effect of sample size ("us"=μsec).

6.1.2 *Effect of sample size*

We were concerned that Figure 7 tells an incomplete story; as the sample size is increased, Freq would eventually win out simply by "rote learning", i.e., placing syncs at frequent training features would also capture most of the test probes. As Figure 8 shows, this is not the case. This is because of the typical heavy-tail behavior of feature hits: as we increase the sample size, we continue to get feature hits in the test fold that were never seen in the train fold, at a rate significant enough to favor Equi and DynProg over Freq. (EquiAndFreq runs were discontinued because they were uninteresting: performance was interpolated between Freq and Equi throughout.)

6.1.3 *Outer policies compared*

Keeping other options at their best values, Figure 9 compares outer policies. As can be seen, Bit is worse than SqrtHitBit, which shows that our bulk model at the outer level does capture some essence of the problem. HitBit is even worse than bit, most likely because feature hits are correlated with leaf bits, and HitBit "double-counts" this effect. Overall, the best outer+inner policy combination can be **six times** faster than the worst, so it is important to choose well.

6.1.4 *Feature renumbering*

Figure 10 shows the effect of one global permutation of feature IDs over all leaves on the size of the RAM LFEM buffer. As commented in Section 4.5, the default feature ID order is far from random. As a baseline, we purposely assigned random IDs—this resulted in a considerable increase in the LFEM buffer size, as expected. We also tried the other policies proposed in Section 4.5. FeatHits reordering reduces LFEM buffer size by over 430 MB. In contrast, the permutation itself costs only 30 MB.

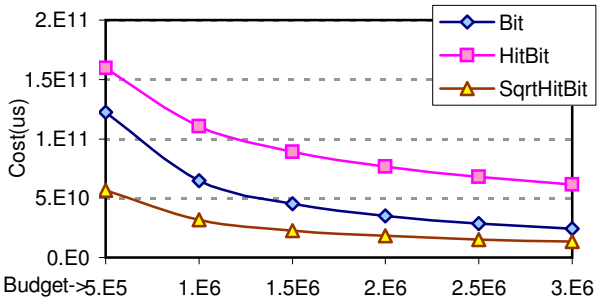


Figure 9: Outer policies compared ("us"=μsec).

Feature ID order	LFEM bytes
Random	1,629,356,922
Default (first encounter)	1,553,006,658
FeatHits: decr. global feature hits	1,123,348,583
Global $\frac{\text{feature hits}}{\text{number of bits}}$, decreasing	1,408,519,022

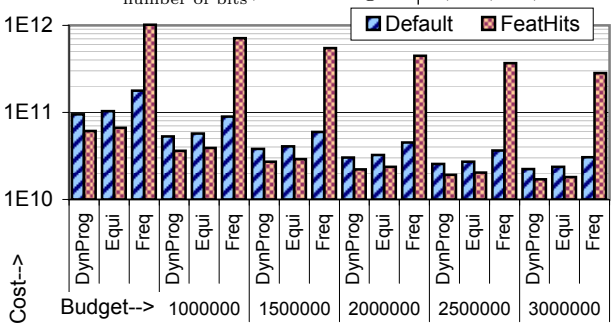


Figure 10: Effect of feature renumbering.

Figure 10 also shows the effect on annotation cost (time). While DynProg and Equi benefit from FeatHit permutation (between 13 and 35%), Freq suffers greatly (almost 10× slowdown). Closer scrutiny of this curious effect showed that the global FeatHits ordering has excellent agreement with feature frequencies within some of the costliest leaves. Therefore, after reordering, Freq pushes all syncs to the very left end of the feature range. Though this results in decreased scan cost for some features, the collective adverse effect of scans to heavy tail features all over the range predominates. By design, Equi and DynProg are immune to this problem.

6.1.5 *Accuracy of estimated cost*

Measured CPU time (y) and modeled cost (x) are in excellent agreement with the regression $y = 1.0645x$ having an R^2 coefficient of 0.9895 (1 means perfect linear fit). Points were chosen from a wide variety of inner and outer policies and sync budgets.

6.1.6 *Speed comparison against other systems*

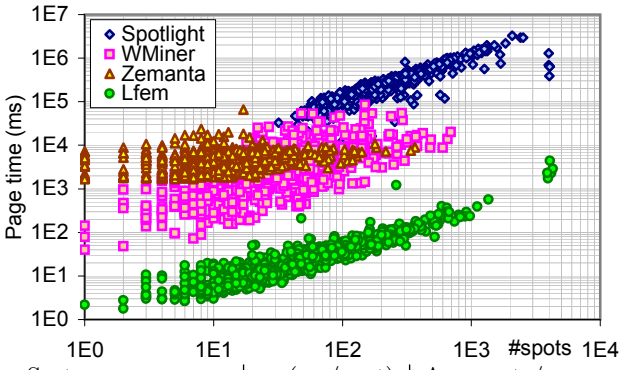
Other than SemTag, no other system [24, 12, 26, 20, 17, 16] reported on running time or memory requirements. SemTag [13] spotted 156 documents per second and disambiguated 1,200–3,000 windows per second on one core, but these numbers are on an incomparable platform.

At the time of writing, three well-known systems were readily available to compare against ours: DBPedia Spotlight, Wikipedia Miner [25], and Zemanta. All the systems have competitive accuracy; here we focus on scalability.

We annotated 1000 documents d, sampled from Wikipedia, of diverse sizes (number of tokens t_d) and measured the number of annotated spots s_d and the time y_d taken. (For Wikipedia Miner, we disabled the initial caching as it otherwise took 2–3 hours to start up. The comparison with Zemanta may not be entirely fair because its catalog extends beyond Wikipedia.)

We fitted a constrained linear regression $y_d = \tau_0 + \tau_1 t_d + \tau_2 s_d$ where all $\tau_i \geq 0$ and $\tau_1 \leq \tau_2$ (tokens not in a spot should cost less). τ_0 captures a per-document overhead (including network latency for online services like Zemanta). Although not all systems gave great fit using linear regression, τ_2 gave a very good idea of each annotator's throughput.

Figure 11 shows a scatter plot of annotation time against the number of annotations produced, and also tabulates τ_2 estimates. LFEM is **16–263 times faster** than other popular systems; thanks entirely to syncs (see last row).



System	τ_2 (ms/spot)	Avg spots/page
DBPedia Spotlight	158	216
Wikipedia Miner	21	48
Zemanta	9.5	16
LFEM	**0.6**	111
LFEM, *no sync*	*42*	-

Figure 11: Throughput of our and other systems.

6.2 SIP index performance

When we proposed EntSIP and DictSIP (Section 5), we had no way to predict which would be better. Figure 12 shows that DictSIP is more compact despite spending space on per-document entity dictionaries. The reason is clear from the cost break-up: a unified posting list reduces gap-gamma bits.

Description	EntSIP	DictSip
Document blocks	117362457	117362457
Total # postings	477450617	477450617
Long+short entity ID bits	12461668521	14800482316
Sub-block delimiter bits	1170992844	0
Gap and span width bits	8638890811	5877067785
Total of above bits	22271552176	20677550101
Average gap	628.8	153.9

Figure 12: EntSIP vs. DictSIP — DictSIP wins.

Figure 13 compares DictSIP against using long (32-bit) entity IDs, and using Lucene's *payload* hook to pack the short entity ID. Long entity IDs take **43%** more bits. As indices steadily move to RAM, this represents significant savings. Lucene payloads are multiples of a byte, which makes it **27%** more expensive than DictSIP. We also tried [7] to use Indri but its inlined annotations can support at most 4000 or so entities plus types, because of an internal BTree restriction.

Figure 14 compares DictSIP with collocation indices. This experiment used a much smaller catalog to keep collocation index sizes in control. We also used a recall-precision knob to control the annotation density per page. Clearly, collocation indices are impractical for large catalogs that we critically need (Section 2)—they are **45–60 times** the size of our DictSIP index, which would overflow our system.

Finally, Figure 15 compares index decompression and scan performance against a plain MG4J [5] text index. Numbers of postings are comparable. DictSIP needs 23% more space

Entity coding option	Space (bits)
DictSIP	177251981035
DictSIP using Lucene payload	224801570189
Inlining long entity IDs	252246867520

Figure 13: Benefits of DictSIP's compact dictionary.

Annotation Density	0.068	0.150	0.299	0.438
Collocation	18.110	18.077	29.000	42.450
DictSIP type	0.290	0.285	0.658	0.828

Figure 14: DictSIP compared with collocation index. About 2M entities and only 617 types were indexed.

Description	MG4J	DictSIP
Postings	8,356,750,159	8,486,091,421
Index shard size	48 GB	59 GB
Doc blocks scanned/sec	0.94 M	1.7 M
Posts scanned/sec	2.2 M	4.97 M
Disk read rate	10–14 MB/s	45 MB/s
Disk read rate (`dd bs=1M`)	105 MB/s	105 MB/s
CPU core utilization	100%	75–85%

Figure 15: Index scan+decompression performance.

per posting to store additional fields, but it can decompress and scan more postings and document blocks per second. With about 50M documents per shard, for typical TREC entity queries (`ilps.science.uva.nl/trec-entity`), we can retrieve all snippets containing entities of a given type together with keywords in 1–5 seconds.

7. CONCLUSION

We presented a fast, Web-scale annotator and annotation indexing system. The two key contributions are 1. a generic compression scheme for fast, compact, in-memory disambiguation data structures and 2. a compact posting list design for entity and type indices. Investigating Golomb, delta and other codes may give additional gains. Recent techniques [20, 17, 16] go beyond local signals, and it would be of interest to support such collective algorithms by extending our data structures. Supplementary material on our project is available at `soumen.cse.iitb.ac.in/doc/CSAW`.

Acknowledgment. Thanks to Natassa Ailamaki for vertical database references and Sebastiano Vigna for much help with MG4J.

8. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD Conference*, pages 671–682. ACM, 2006.

[2] S. Agrawal, K. Chakrabarti, S. Chaudhuri, V. Ganti, A. C. König, and D. Xin. Exploiting web search engines to search structured databases. In *WWW Conference*, pages 501–510, 2009.

[3] K. Balog, L. Azzopardi, and M. de Rijke. A language modeling framework for expert finding. *Information Processing and Management*, 45(1):1–19, 2009.

[4] P. Boldi and S. Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. In *String Processing and Information Retrieval*, volume 3772 of *LNCS*, pages 25–28. Springer, 2005.

[5] P. Boldi and S. Vigna. MG4J at TREC 2005. In E. M. Voorhees and L. P. Buckland, editors, *TREC*, number SP 500-266 in Special Publications. NIST, 2005.

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

[7] S. Chakrabarti, D. Sane, and G. Ramakrishnan. Web-scale entity-relation search architecture (poster). In *WWW Conference*, pages 21–22, 2011.

[8] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE*, page 28, 2006.

[9] T. Cheng and K. C.-C. Chang. Beyond pages: supporting efficient, scalable entity search with dual-inversion index. In *EDBT*, pages 15–26. ACM, 2010.

[10] T. Cheng, X. Yan, and K. C. Chang. EntityRank: Searching entities directly and holistically. In *VLDB Conference*, pages 387–398, Sept. 2007.

[11] F. Chierichetti, S. Lattanzi, F. Mari, and A. Panconesi. On placing skips optimally in expectation. In *WSDM Conference*, pages 15–24, 2008.

[12] S. Cucerzan. Large-scale named entity disambiguation based on Wikipedia data. In *EMNLP Conference*, pages 708–716, 2007.

[13] S. Dill et al. SemTag and Seeker: Bootstrapping the semantic Web via automated semantic annotation. In *WWW Conference*, pages 178–186, 2003.

[14] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In *WWW Conference*, pages 311–320, 2010.

[15] R. V. Guha and R. McCool. TAP: A semantic web test-bed. *Journal of Web Semantics*, 1(1):81–87, 2003.

[16] X. Han, L. Sun, and J. Zhao. Collective entity linking in Web text: A graph-based method. In *SIGIR Conference*, pages 765–774, 2011.

[17] J. Hoffart et al. Robust disambiguation of named entities in text. In *EMNLP Conference*, pages 782–792, Edinburgh, Scotland, UK, July 2011. SIGDAT.

[18] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal of Applied Mathematics*, 21(4):514–532, 1971.

[19] G. Kasneci, F. M. Suchanek, G. Ifrim, S. Elbassuoni, M. Ramanath, and G. Weikum. NAGA: harvesting, searching and ranking knowledge. In *SIGMOD Conference*, pages 1285–1288. ACM, 2008.

[20] S. Kulkarni, A. Singh, G. Ramakrishnan, and S. Chakrabarti. Collective annotation of Wikipedia entities in Web text. In *SIGKDD Conference*, pages 457–466, 2009.

[21] X. Li, C. Li, and C. Yu. EntityEngine: Answering entity-relationship queries using shallow semantics. In *CIKM*, Oct. 2010. (demo).

[22] C. Martinez and S. Roura. Optimal and nearly optimal static weighted skip lists. Technical Report LSI-95-34-R, Universitat Politècnica de Catalunya, 1995.

[23] P. N. Mendes, M. Jakob, A. García-Silva, and C. Bizer. DBpedia Spotlight: Shedding light on the Web of documents. In *7th International Conference on Semantic Systems*, pages 1–8, 2011.

[24] R. Mihalcea and A. Csomai. Wikify!: linking documents to encyclopedic knowledge. In *CIKM*, pages 233–242, 2007.

[25] D. Milne. An open-source toolkit for mining Wikipedia (`wikipedia-miner.sourceforge.net/publications.htm`). To be announced, 2009.

[26] D. Milne and I. H. Witten. Learning to link with Wikipedia. In *CIKM*, pages 509–518, 2008.

[27] S. Sarawagi. Information extraction. *FnT Databases*, 1(3), 2008.

[28] V. R. Shanks, H. E. Williams, and A. Cannane. Indexing for fast categorisation. In *Australasian Computer Science Conference*, volume 16 of *ACSC26*, pages 119–127, Darlinghurst, Australia, 2003.

[29] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A core of semantic knowledge unifying WordNet and Wikipedia. In *WWW Conference*, pages 697–706. ACM Press, 2007.

[30] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan-Kaufmann, May 1999.

[31] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, pages 59–70, 2006.