Contextual Indexing and Joining: Supporting Efficient, Scalable Entity Search

Tao Cheng and Kevin Chen-Chuan Chang
Computer Science Department
University of Illinois at Urbana-Champaign
{tcheng3,kcchang}@cs.uiuc.edu

ABSTRACT
As the Web has evolved into an entity abundant repository, with the standard “page view”, current search engines are becoming increasingly inadequate for a wide range of query tasks. Entity search, a significant departure from document retrieval, finds fine granularity information, i.e., entities, embedded in documents directly and holistically across the whole collection. Essentially, entity search is to find matching entities by context patterns from each document and to aggregate them across documents for ranking. This text-based pattern matching suggests that standard inverted lists-based query processing can be applied. However, this baseline is limited in both efficiency, due to long entity lists, and scalability, due to cross-document aggregation. To enhance efficiency, we propose “contextual index”, an index that materializes pre-joins, to eliminate unnecessary index reading and reduce online matching. To improve scalability, we propose “entity-space” partitioning, so that answer subspaces can be aggregated locally. We reason our design rationale from both the functional and the operational definition of entity search, and show that they consistently reach our framework.

We evaluate the indexing (contextual indexing) and parallel query processing (contextual joining) framework over a 2TB real Web corpus with systematic benchmark query sets. Experiments show that our scheme can speed up query processing by, in average, two order of magnitude over the baseline.

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). 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”, as many recent efforts also aim at (e.g., [4, 5, 17, 15, 13, 12, 1, 2, 3, 14, 19]), we have proposed the notion of entity search over a corpus of text documents (such as the Web), in terms of its ranking function [7] and its applications in information integration [6]. Upon this foundation, to realize ad-hoc entity search over a large corpus, this paper studies the key challenges of query processing.

As the Web is rich with various type of data, users are often looking for specific “fine grained” information, or data objects of specific types, each of which we call an entity. The notion of entity can broadly refer to anything that can be reasonably recognized from the text corpus, either straightforwardly or with sophisticated techniques, often with uncertainty (see Section 2). To motivate, consider user Amy: She may be looking for the “phone number” of say, Amazon.com’s customer service? To apply for graduate school, how can she find the list of “professors” in the database area? 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? Or, to buy Shakespeare’s Hamlet, how can she find the “prices” and “cover images” of available choices from, say, Borders.com and BN.com?

In these scenarios, like every user in many similar situations, Amy is looking for particular entities of information, e.g., a phone number, a book cover image, a PDF, a PPT, a name, a date, an email address, etc. We thus aim at supporting entity search, to directly find matching entities across as many pages as they may occur. To illustrate, our scenarios will lead to the following queries:

Q1: ow20 (amazon service #phone)
Q2: ( #professor #university #research = “database”)
Q3: ow (sigmod 2006 #pdf_file #ppt_file)
Q4: (%title = “hamlet” #image #price)

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 surrounding "context" with a right answer. To distinguish entities and keywords, we use a prefix #, e.g., #phone for the phone entity. Each query is thus essentially 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). We may also omit the window size (e.g., Q3, which is default to 100 words window) or even the entire 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.

Second, as output, users will directly get their desired entities. 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. Figure 1 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 1 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. 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 1 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 as engine 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.

Toward supporting such entity search, there are several open issues we must address. To begin with, how to effectively score and rank entities? As we studied in [7], there are several unique requirements. 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. As a foundation, we have developed EntityRank and demonstrated its effectiveness in [7].

This paper focuses on the ensuing challenge for online entity search, how to support efficient and scalable query processing? While we started with extending standard document search in a “natural” way (Section 2), which indexes entities in the same way as keywords in inverted lists, we found this baseline, called Basic, quite inadequate: First, Basic is inefficient because of the lengthy lists of entities. Note that each entity (e.g., #phone) is not a unique literal value, unlike a keyword; instead, it is a large set of instance values (e.g., the numerous phone numbers on the Web). The extremely-long inverted lists of entities thus dominate and slow down query processing. Second, Basic is non-scalable, because it parallelizes by partitioning the document space. Such partitioning, while natural for page search, does not amortize computation to the local nodes in a cluster, and thus leaves the central node as the bottleneck. This paper proposes our solutions, which addresses the issue of 1) index design, for which we develop contextual index, and 2) data parallelization, for which we develop partitioning by entity space.

**Figure 1:** Query Results of Q1 and Q3

<table>
<thead>
<tr>
<th>rank</th>
<th>phone number</th>
<th>score</th>
<th>urls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>800-201-7575</td>
<td>0.9</td>
<td>amazon.com/support.htm</td>
</tr>
<tr>
<td>2</td>
<td>800-322-9266</td>
<td>0.3</td>
<td>myblog.org/shopping</td>
</tr>
<tr>
<td>3</td>
<td>800-342-5283</td>
<td>0.6</td>
<td>rzy.com</td>
</tr>
<tr>
<td>4</td>
<td>206-346-2992</td>
<td>0.2</td>
<td>hp.com</td>
</tr>
</tbody>
</table>

2. THE PROBLEM: ENTITY SEARCH

We now define entity search, and identify its challenges in query processing. We will discuss from dual perspectives: First, to see the “semantics” of entity search, we will examine the functional perspective, in terms of how the output is related to the input (Section 2.1). Second, to understand its computation, the focus of this paper, we will develop the operational perspective (Section 2.2). Together, they will lead us to identifying the challenges in query processing (Section 2.3). The dual perspectives will further inspire consistent insights for a solution, which Section 3 will present.

For our discussion, we will use Figure 2 as a running example, which we call the YellowPage scenario, as it provides search for contact information #phone and #email. As a toy dataset, the corpus D has 100 documents D = {d1, ..., d100}; we show three documents d9, d20, d35 as examples. (The scenario is indeed our experimental setting, in which we indexed 90+ million pages.) We will also assume Q1 (Section 1) as the query.

**Figure 2:** A running example: YellowPage.

Our results show that contextual indexing and its data parallelization is quite efficient: In our prototype indexing 2 TB of real Web corpus (with 90+ million pages), across a cluster of 34 nodes, for a YellowPage setting (similar to that of query Q1), we systematically queried a sample of Fortune-500 companies and SIGMOD 2007 PC members, with varying number of keywords and entities. Our contextual-index approach constantly and significantly outperforms the Basic baseline of current document-based search, with an average of 200 - 500 times speedup, or over two orders of magnitude faster. In terms of absolute time, the speedup reduces response time from, in average, 11.5 seconds (of Basic) to 0.09 second (of our scheme), thus in effect making on-line query processing possible. We note that the speedup is constant across our analysis with respect to the number of keywords, the number of entities, and the “selectivity” of context pre-joining, as Section 5 will report.

We start in Section 2 to formalize entity search and its challenges. Section 3 proposes our solution, for which Section 4 concretely develops its realization. Section 5 reports our prototype system and experiments, and we relate to existing studies in Section 6. **Contributions**

1. As the foundation for abstracting the computation of entity search, we identify its operational definition. (Section 2)
2. We develop index design and data parallelization for query processing, reasoning from both the functional and operational perspectives of entity search. (Section 3, 4)
3. We have implemented the contextual index-based entity searcher in an online prototype with real Web corpus, and demonstrated the effectiveness and efficiency of entity search.

2.1 Entity Search: Functional Definition

To understand its semantics, we define entity search from a functional perspective, i.e., input, output, and how they are related.
Data Model: To view the Web as a repository to search over, in contrast to page-oriented search, where the Web is a set of documents (or pages) \( D = \{d_1, \ldots, d_n\} \), we take an entity view, which considers the Web as primarily a repository of entities (which appear over pages): \( E = \{E_1, \ldots, E_N\} \). The set of entity types \( E_i \) to support depends on the search applications (much like the “schema” of a database application defines attributes to use). E.g., to support Scenario 1 (Section 1), i.e., our YellowPage setting (Figure 2), the system might be constructed with entities \( E = \{\#phone, \#email\} \). In [7], we presented several applications of entity search with various schemas.

Each entity \( E_i \) is a set of entity instances \( \{e_{i1}, \ldots, e_{ih}\} \) that are extracted from the corpus, i.e., literal values of entity type \( E_i \) that occur somewhere in some \( d \in D \). 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 = \{“800-201-7575”, “244-2919”, “(217) 344-9788”, . . . \}. As Figure 2 shows, for YellowPage, \#phone is a set of 100 distinctive phone entity instances \( \{p_1, \ldots, p_{100}\} \) and \#email is a set of 100 distinctive email entity instances \( \{e_1, \ldots, e_{100}\} \).

These entities are recognized offline, and their instances are indexed for efficient search– which is the focus of this paper. For entity extraction, we can adopt a range of existing techniques: from simple file types (e.g., \#pdf, \#ppt), dictionaries (e.g., \#state, \#professor, \#senator), pattern matching (e.g., \#phone, \#email), to state-of-the-art named-entity taggers (e.g., entityname, \#organization).

Our system description [8] discusses our current implementation of entity extraction.

To enable query matching in search, for each entity instance \( e \) (of type \( E_i \), e.g. a particular \#phone (say, 217-321-1234), as it may occur many times, we will record all its occurrences across documents in \( D \). At offline indexing time, by scanning through \( D \), each occurrence of an entity instance will be recognized by entity extraction (as just explained). For each occurrence \( o \), we record certain properties (or “features”) that characterize what \( o \) is, in order for online matching with queries. While the exact choice of features depend on the ranking algorithm, they must capture the appearance, confidence, and position of each occurrence. For concrete discussion, without loss of generality, we assume the following simple properties, which our ranking model EntityRank [7] actually uses.

- Where does \( o \) occur? As the position, \( o.\text{docid} \) is a document id and \( o.\text{attpos} \) is a word offset of occurrence.
- What instance does \( o \) represent? As the instance, \( o.\text{inst} \) is of the form \( E_i.e \) indicating an instance id \( e \) with respect to type \( E_i \).
- How certain is the extraction? Confidence \( o.\text{conf} \) is an probability estimation given by the entity extractor.

Our entity view thus considers the extraction of \( E_i \) over \( D \) as a relation (i.e., a set of tuples, each with the above fields) of all \( E_i \) occurrences. (As we will see in the operational perspective, considering the entity view as “relational” enables us to define operations algebraically.) Each \( E_i \) thus induces an occurrence relation \( I(E_i) \) as follows, e.g., \( I(\#phone) \) for YellowPage, as in Figure 2.

\[
I(E_i) = \{o.\text{docid}, o.\text{pos}, o.\text{inst}, o.\text{conf}\}, s.t. o \text{ occurs in } D, o.\text{inst} \in E_i, \}
\]

Meanwhile, keywords remain essential, as they are used to match with entities, much like their role in document search. Like entities, each keyword \( k_j \), say “amazon,” can occur many times in \( D \), which we also record in a similar occurrence relation. However, unlike entities, each occurrence \( x \) of \( k_j \) is a literal value of itself without multiple different instances, and it can be recognized without uncertainty. Thus, the occurrence relation of \( k_j \) is of a simpler form, e.g., \( I(\text{amazon}) \) for YellowPage (Figure 2).

\[
I(k_j) = \{\kappa(\text{docid}, \text{pos}), s.t. o \text{ occurs in } D\}
\]

Figure 3: Entity Search: Functional Definition.

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

Search Problem: By independently extracting each entity, we have transformed the corpus \( D \) into our entity view, as a searchable collection of entities \( E \). The search problem is thus, given a query with keywords (e.g., for Q1: “amazon service”) and desired entities (e.g., \#phone, \#email or both), to find matching instances, such that the association of these instances and keywords are evident from the corpus. E.g., for Q1, we will find the association (“amazon service”, \#phone) or (“amazon service”, \#phone, \#email) if both entities are specified. Supporting such online matching and association is exactly the challenge (and usefulness) of entity search.

We now state the entity search problem, as Figure 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 \( D \) by certain patterns. We note that, as Section 1 motivated, 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 and Q3) or implicitly assume the system default pattern (e.g., Q2 and Q4).

Second, for output, the results are a ranked list of \( m \)-ary entity tuples, each of the form \( t = \{e_1, \ldots, e_m\} \), i.e., a combined instance of each \( e_i \) as an instance of entity \( E_i \) in the query. A 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 \( \text{Score}(q,t) \).

Overall, the entity search problem is thus, given \( q \), to find from the search space of \( t \in E_1 \times \ldots \times E_m \), the matching tuples in the ranked order by \( \text{Score}(q,t) \).

Although the detail may vary, since we are evaluating \( q \) over a corpus \( D \) of documents \( d_1, \ldots, d_n \), a scoring function should capture how \( t = \{e_1, \ldots, e_m\} \) appears, by the desired tuple pattern \( \alpha \), by matching all its occurrences across all documents. Given \( o_i \) as an occurrence of each \( e_i \) (i.e., \( o_i.\text{inst} = e_i \)) in some document \( d \), each occurrence \( \alpha, \text{docid} = d \) and \( k_j \), as an occurrence of keyword \( k_j \), they form a combined tuple-occurrence \( \omega = (o_1, \ldots, o_m, k_1, \ldots, k_l) \). Many such occurrences will appear in various documents in \( D \). Thus a scoring function is generally of the form \( \text{Score}(q,t) = \sum_{\omega \in (o_1, \ldots, o_m, k_1, \ldots, k_l)} G_{\omega}(o_1, \ldots, o_m, k_1, \ldots, k_l) \).

1. \( L_o \): local recognition. For each tuple-occurrence \( \omega, L \) determines a “local score” of how \( \omega \) matches \( \alpha \).
2. \( G \): global aggregation. Across all occurrences \( \omega \) in all documents in \( D \), \( G \) aggregates them globally into the total score.

While the effectiveness of entity search hinges on the scoring function, this paper focuses on the efficiency of query processing, assuming functions of the above general form. For our concrete discussion, let’s assume a simplistic scoring function, \( \text{BinarySum} \), as our running example. In [7], we developed an Impression Model, and consequently the EntityRank function, which simulates a ran-
The Baseline: Basic. Can entity search be implemented similarly to document search? As both are dealing with text, entity search can indeed use a similar framework, which we call the Basic baseline, with inverted lists and sort-merge joins. Referring to Figure 4, first, each occurrence relation \( I(E_i) \) and \( I(k_j) \) can be stored and sorted in the same way as standard inverted lists; the only difference is that, for entities, each posting (or occurrence) \( o \) has more properties, i.e., \( \langle \text{docid}, \text{pos}, \text{inst}, \text{conf} \rangle \). Second, these lists can then be similarly sort-merged, to compute the context join \( L_a \).

However, while our development started with this simple baseline, to our surprise, we found it quite inadequate: First, it is inefficient: The inverted lists of entities are often extremely long, which dominate and slow down processing significantly. Unlike literal keywords (e.g., “amazon”), each entity \( E_i \) represents a large set of instances, e.g., \#phone contains various phone numbers (true or not) that can be extracted from the Web, and thus its list \( I(\#phone) \) is much longer than that of a typical keyword. Such a long list will entail long index loading time and CPU join time.

Second, it is non-scalable: The requirement group-and-aggregation renders simple corpus partitioning unsuitable for parallelization. It is well known, as some researchers put it, that document search is “embarrassingly parallelizable.” For a cluster of, say, 100 nodes, we may simply partition the corpus \( D \) into 100 sub-corpora, each containing a 1% subset of the documents. Each node will index and search the sub-corpus independently, and the results are simply merged to produce an overall ranked list. Unfortunately, this simple document-based partitioning is, again, not suitable for entity search. The difficulty arises because of \( \mathcal{G} \), the grouping and aggregation by entity instances (operation 2 in Figure 4). With simple corpus partitioning, since the same entity instance can have occurrences from different sub-corpus, \( \mathcal{G} \) can only be performed at a central node, which becomes a bottleneck.

Recognizing the need for a departure from standard document search, we thus aim at developing query processing for entity search, to address the two central issues of efficiency and scalability.

3. OUR PROPOSAL

This section will introduce our high level proposal to deal with the dual challenges of efficiency and scalability from both the functional perspective and the operational perspective. While we try to develop from both perspectives, we will focus more concretely on our proposal using the operational perspective, since it is directly

\[
\begin{align*}
\text{score}(E_1, \ldots, E_m) \cdot \mathcal{G}(\omega) &= \text{score}(I(E_1), \ldots, I(E_m), I(k_1), \ldots, I(k_l)) \\
\text{score}(E_1, \ldots, E_m) \cdot \mathcal{G}(\omega) &= \text{score}(I(E_1), \ldots, I(E_m), I(k_1), \ldots, I(k_l)) \\
\end{align*}
\]

Figure 4: Entity Search: Operational Procedures.
the computation model (4). However, as we will show in this section, the two perspectives lead to the same solution.

We now propose our solution for an entity searcher. As Section 2.1 defines, the challenges of efficiency and scalability arise from two central issues, which our entity searcher must address.

11. **How to index** \( I(E_i) \) and \( I(k_j) \)? In baseline Basic, adapting inverted lists results in long entity lists \( I(E_i) \), which dominate and slow down query processing.

12. **How to partition the corpus?** In the baseline, document-based partitioning renders aggregation a central bottleneck.

Parallel to the dual perspectives, functional and operational, of defining entity search (Section 2), we will also derive from the two aspects, in Section 3.1 and 3.2. It is interesting to note that, from not only the functional nature but also the computation aspect, we obtain the same conclusion.

### 3.1 Functional Perspective

From the functional perspective, we would like to reason from the nature of the search problem, as Section 2.1 defines. Unlike the direct adaptation in Basic, we wish to draw deeper insights from document search, to parallel our design of an entity searcher with that of a document searcher.

As Section 2.1 defines (Figure 3), given query \( \alpha(E_1, \ldots, E_m, k_1, \ldots, k_l) \), an entity searcher must find entity instances \( t(\ldots, e_m, \ldots) \), from the space \( E_1 \times \ldots \times E_m \), that match those keywords in their context (e.g., phone around the mentions of “amazon” and “service”). In contrast, for document search, a similar query finds documents from the space \( D = \{d_1, \ldots, d_n\} \) by matching keywords in their contents (e.g., documents containing “amazon” and “service”). Figure 5 highlights these contrasts in search space and matching pattern. With these contrasts, we can parallel our design with document search, reaching two design principles:

**Principle A1: Indexing by Inverting to Entities via Context.** Observe that, for document search, an inverted list for keyword \( k \) is an inversion from \( k \) to each document \( d \in D \), where \( k \) occurs within its content, or, in short, indexing by inverting to documents via content. Generalizing this observation, as we are now searching for entities as targets, by keywords in their context, our indexes shall be the inversions from keywords to each instance \( e_i \in E_i \), where \( k \) occurs around its context. (How to define this “context”—such as a certain-sized window, is an implementation decision.)

**Principle A2: Parallelizing by Partitioning the Entity Space.** Observe that, for document search, the parallelization scheme naturally partitions the search space \( D \) into disjoint subsets \( D_1, \ldots, D_n \), i.e., \( D_1 \cup \ldots \cup D_n = D \) and \( D_i \cap D_j = \emptyset \). As our search target now is entities, applying the same principle, we should parallelize by partitioning the space of each entity \( E_i \), into non-disjoint subsets \( E_{i1}, \ldots, E_{im} \) s.t. \( E_{i1} \cup \ldots \cup E_{im} = E \) and \( E_{i1} \cap E_{i2} = \emptyset \).

### 3.2 Operational Perspective

Will our conclusion be the same, if we reason from the operational perspective? As Section 2.2 discusses (which Figure 4 summarizes), entity search computationally consists of join, aggregation, and ranking. From the perspective of speeding up and parallelizing this process, we will reach the following two principles:

**Principle B1: Indexing by Pre-Computing Context Joins.** To speed up, as the inefficiency lies in the lengthy entity lists \( E_i \), the remedy is naturally to perform pre-computation, and build the results into indexes. What pre-computation will speed up the context-join \( \eta_{k\alpha}(I(E_1), \ldots, I(E_m), I(k_1), \ldots, I(k_l)) \)? Clearly, any sub-joins will help. E.g., with sub-join \( C_{\#p} = \eta_{\#p}(I(amazon), I(#phone)) \), materialized, for queries like \( Q1 \) we can merge this result with the missing components to produce the overall join, i.e., \( \eta_{\alpha}(I(amazon), I(#service), I(#phone)) = \eta_{\alpha}(C_{\#ap}, I(#service)) \). Why is this more efficient? By pre-joining “amazon” and “#phone”, \( C_{\#ap} \) not only contains all the necessary instances of both, but also prunes out many “unmatched” ones, thus reducing both index loading and joining time. This principle speeds up full joins by materializing some or all parts of it, and build the “sub-joins” into indexes.

Note that, in pre-computation, we are concerned with only matching patterns \( \alpha \); the actual scoring by \( L \) should only be executed in the final full joins. Thus, to simplify discussion, we will only show patterns in joins here. Further, since we want to build materialized sub-joins for as many queries as possible, we should execute pruning by a super pattern (e.g., \( \alpha^* = \#100 \), or within 100-word window of proximity) which will subsume any patterns that the system supports (e.g., \( \alpha = \#20 \) as in \( Q1 \); i.e., \( \alpha = \alpha^* \)). The choice of the super pattern depend on implementation.

To understand the design issues of what sub-joins to materialize, we look further into how they may help with query processing: Let’s denote a (full or partial) context-join as \( \eta_{\alpha}(I) \), where \( I \) is the set of (entities or keyword) occurrence relations covered in the join; i.e., \( C_{\#ap} \) has \( I = \{I(amazon), I(#phone)\} \). Note that the individual occurrence relation, such as \( I(amazon) \), is thus a special case where \( |I| = 1 \), representing a basic inverted list. We can state how sub-joins contribute as follows:

\[
\begin{align*}
\eta_{\alpha}(I) &= \eta_{\alpha}\left(\eta_{\alpha^*}(I_1), \ldots, \eta_{\alpha^*}(I_n)\right), \\
&\text{if } I = \cup\{I_1, \ldots, I_n\}.
\end{align*}
\]

Concretely, we propose to pre-join only pairs of keywords and entities: We will build contextual index, by context-joining every pair of one keyword \( k_j \) and one entity \( E_i \):

\[
C(k_j, E_i) = \eta_{\alpha^*}\left(I(k_j), I(E_i)\right) \quad \forall E_i, \forall k_j.
\]

Remark: Principle B1 is consistent with A1. Observe that, each contextual index \( C(k_j, E_i) \) associates \( k_j \) with \( E_i \), when they together match a context pattern \( \alpha^* \). From the standpoint of keyword \( k_j \), the contextual index is to find all entity instances, in whose context \( k_j \) appears— Thus clearly, it is indeed an inversion from \( k_j \) to entities via a context pattern.

**Principle B2: Parallelizing by Partitioning along Groups.** To parallelize the join-aggregation-ranking process in Figure 4, since the final ranking must be performed to the overall results, it must
remain at the central node, and thus our objective is to “push down” the aggregation \(G\) to be executed at each local node. As \(G\) performs aggregation (by function \(G\)) for each group of the same entity instances, to push it down to a local node, we must make sure that each node has an entire group for \((e_1, \ldots, e_n)\). Thus, this principle suggests partitioning along groups.

Since we have proposed using contextual indexes, as Eq. 4 defines, our issue now is how to partition these lists. (These lists will be stored by extending the structure of inverted lists; see Section 4.) For a contextual index \(C = C(k_j, E_i)\), how to partition it? To partition along the groups, we make sure the same instances of \(E_i\) will be allocated at the same local node, which means we must divide \(E_i\) into subsets. Specifically, we partition \(E_i\) to \(n\) nodes, i.e., \(E_i = \cup \{E_i^1, \ldots, E_i^n\}\), and consequently the partition of contextual index \(C\) follows: We can compute the sub-index \(C^*\) by pre-joining only the entity subset \(E_i^j\), or simply computing \(C\) once and “projecting” it to the subset, as follows:

\[
C^* = C(k_j, E_i^j) = \forall v_o \cdot I(k_j), I(E_i^j)] = C[E_i^j]
\]

(4)

This partition, by dividing contextual indexes, is best suited for the common cases of simple queries. For queries with one entity (e.g., \(Q_1\), say \(E_i\)), this scheme gives highly parallel processing. Since each tuple \(t\) contains just an instance of \(E_i\) (i.e., \(t = (e_i)\), for \(e_i \in E_i\), the corresponding group is fully contained in the local node where \(e_i\) is allocated to, and thus the aggregation \(G\) can indeed be pushed down fully, leaving only ranking at the central node. Section 4 will present the detail. For queries with multiple entities (e.g., \#phone and \#email), as each group is for a composite instance \((e_1, \ldots, e_n)\), their groups will form at the central node. However, the lists to be processed will be significantly reduced at each local node, before reaching the central node, which will still yield significant speedup compared to Basic (as Section 5 will show).

Remark: Principle B2 is consistent with A2. The grouping in B2 by instance values of \(E_i\) is exactly the entity space of \(E_i\) (and the composite situations of \(E_i \times \cdots \times E_m\).

4. REALIZATION

To concretely realize our proposal in section 3, namely principle B1 and B2 as section 3.2 concluded, we will focus on index design and parallel query processing respectively. Section 4.1 will discuss our contextual index design for dealing with the efficiency challenge and section 4.2 will discuss the overall query processing to deal with the scalability challenge.

4.1 Contextual Index

Let’s reuse query \(Q_1\), finding Amazon’s customer service phone number, as our running example query.

Baseline Implementation: In our Basic approach, we index entities the same way as we index keyword using the commonly used data structure: inverted index. A keyword inverted index records the occurrence table of the keyword \(I(k_j)\) in an ordered list sorted by document id. Similarly for a specific entity type \(E_i\), all its occurrences can be stored in an inverted index, where entries are ordered by document id. This is the actual realization of occurrence relation \(I(E_i)\) which we introduced in Section 2. To optimize and save space, all occurrences that appear in the same document can share one docid.

<table>
<thead>
<tr>
<th>Query</th>
<th>Docid</th>
<th>Docid</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I(k))</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>(I(e))</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>(I(#phone))</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>(I(#email))</td>
<td>27</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Figure 6: Inverted Index Example

Figure 6 shows the layout of the inverted lists \(I_e\), \(I_s\), and \(I_{#p}\) for keywords “amazon”, “service”, and entities “#phone”, “#email” respectively. As we can see, the layout of an entity list resembles that of keywords, except that for each occurrence, instance id and confidence information are stored in addition to the position information.

The actual execution of entity search is essentially for each keyword and entity specified in the query, load their inverted lists into memory; advance all the lists in parallel checking intersecting documents; for each document in the intersection of all lists, using the specified matching pattern to instantiate tuples and calculate their local scores; calculate the final score for each entity tuple by aggregating all its local scores.

Example 2 (Answering Q1 using Inverted Index): Now let’s execute query \(Q_1\) using the inverted index in Figure 6 with the BinarySum scoring function.

We will first load the three lists \(I_e\), \(I_s\), \(I_{#p}\), and \(I_{#e}\) from disk and then advance these lists in parallel. We will find document \(d_0\) in the intersection of all the lists. Phone instance \(p_{10}\) in this document is a matching tuple as the positions of keywords and entity (17, 18, 23 respectively) match the specified pattern. The BinarySum measure used will report local score 1 for this tuple. Notice instance \(p_{10}\) in the document won’t be matched as it falls out of the window of size 20. Similarly, we will report the matching of instance \(p_{96}\) with local score 1 in document \(d_0\), and \(p_{96}\) with local score 1 in document \(d_{97}\).

Efficiency Problems: Although the whole matching process seems to be straightforward, there are obvious redundant operations. First, many document entries that will not produce any matchings are loaded and checked (e.g., document entry \(d_2\) in the “amazon” inverted list, etc). Most of the document entries, noted using “...”, in the “#phohe” list do not need to be loaded. This could significantly save index loading time. Moreover, document intersection check can also be avoided on such document entries. Second, many within-document pattern matching operations are also redundant as they will not produce any matchings (e.g., instance \(p_{10}\) in document \(d_0\).

Our Solution: These aspects motivate the need of pre-computation to reduce unnecessary online computation, as we previously revealed in principle B1 in section 2.2. Our span model discussed in [7] restricts the pattern matching within a maximal window size of 100. Therefore, the super pattern \(\alpha\) in our implementation essentially requires to record all the joins between a keyword and an entity within window size 100.

![Figure 7: Contextual Index Example](image)

Figure 7 shows the layout of the contextual index built from the inverted index in Figure 6. As we can see, each entry in the contextual index records one possible matching between a keyword and an entity, where the position of the keyword, position of the entity, instance id and confidence of the instance are stored (e.g., \(d_0\) : \([17, 23, p_{10}, 0.8]\)).

Example 3 (Answering Q1 using Contextual Index): To answer the same query \(Q_1\) using the contextual index shown in Figure 7, similarly using inverted index, we will also first load the relevant index lists into memory and then walk through the loaded lists in parallel to perform sort-merge join. In this specific example, we will load the contextual index of \(C_{#pp}\) and \(C_{#pp}\) and walk
through the two lists in parallel to find matchings. The exact same results will be produced. As we can see, the unnecessary loading and checking of entries \(d_2, d_3, d_4\) in \(L_1\), entries \(d_5, d_6, d_7\) in \(L_2\), and many entries in \(I_{\text{doc}}\) (abbreviated by ‘...’) are voided. Furthermore, unnecessary matchings within document \(d_8\) and possible many more within other documents are also avoided.

\[
C(k, E):
\begin{array}{ccccccc}
\hline
\text{docid} & \text{entity} & \text{conf} & \text{entity} & \text{inst} & \text{entity} & \text{pos} & \text{keyword} & \text{pos} \\
\hline
\end{array}
\]

Figure 8: Contextual Index Structure

Finally, let’s generalize and describe the structure of contextual index. We show our design of the contextual index in Figure 8. For each keyword \(k_i\) and entity type \(E_j\) in the system, we will build such a contextual index. Such an index contains a series of docids (the ID of a document) and contextual infos (the context information with in the document). The bottom level box shows in detail the structure of contextual info, a series of matching occurrences of the keyword and entity. Each matching is described by keyword pos (the position of the keyword), entity pos (the position of the entity instance) entity inst (the entity instance) and entity conf (the extraction confidence of the entity instance).

4.2 Parallel Query Processing

We now deal with another important problem for supporting entity search. How can we build the search system, such that it can scale with the size of the data?

Baseline Implementation: Traditional search engines are well-known for their ability to process large-scale datasets. The most common way to scale up is to use many processing units and partition the dataset into subsets. This is due to the nature of the computing, where processing of one document is independent of other documents.

To answer the same query \(Q_1\), we will execute the query on each of the local nodes. Local node 1 will produce two matchings for phone instance \(p_8\) matched in document \(d_5\) and \(p_9\) matched in document \(d_8\) by joining sublists \(I^1_{p_8}\) and \(I^1_{p_9}\). Local node 9 will produce one matching for phone instance \(p_8\) matched in document \(d_9\). Other local nodes will not produce any matchings. All the local matchings have to be sent up to the global processing layer, where results are aggregated and finally sorted.

Scalability Problems: While the Basic approach does distribute the sort-merge join operations across all the local nodes, it has its drawbacks: First, this approach requires a global processing node to do the hard work of grouping, aggregating and ranking. For queries that generate many matchings, this process could be time consuming; Second, this architecture also implies that the global processing node has to maintain the information of all the entities, such as the mapping from instance id to actual instance string value. Overall, this document partition based system architecture requires a heavy central node for both storing much meta information and performing extensive computation. Is it possible to distribute as much meta information and computation as possible down to the local processing units as well?

Our Solution: As we pointed out in principle B2 in section 2.2, we propose to partition the dataset on the entity space to facilitate grouping of results. Using the same 10 local processing units, we could partition dataset such that local node 1 is responsible for phone entity instances \(p_1, \ldots, p_{10}\) and email entity instance \(e_1, \ldots, e_{10}\). This partition scheme on the contextual index, implies that we have to split the contextual index along the entity space. Take the contextual list \(C_{\alpha^{p}}\) as an example. This list will be split into two nonempty sublists. Local node 1 will hold sublist \(C_{\alpha^{p}}\) with entries \(d_5: [17, 23, p_8, 0.8]\) and \(d_9: [45, 50, p_8, 0.8]\) and local node 9 will hold sublist \(C_{\alpha^{p}}\) with entry \(d_8: [34, 45, p_8, 0.8]\). The metadata of the entities could also be distributed across the local nodes in this fashion.

Figure 10 shows this entity partition based architecture, where the grouping and aggregating operations can be pushed down to the local layer and the global layer only has to perform ranking.

Example 4 (Inverted Index Partitioned on Document Space): Use the same example we set up in the beginning of this section. Assuming we have 10 local processing units, we can partition the dataset containing 100 document into 10 subset, each containing 10 documents (i.e., the first node will contain documents \(d_1, \ldots, d_{10}\), so on and so forth). This implies the inverted index will be partitioned into sublists. For instance, \(L_1\) in Figure 6 will be partitioned into 10 sublists, \(I^1_{d_1}, \ldots, I^1_{d_{10}}\) respectively. Figure 9 shows this document partition based architecture, where the local layer performs the sort-merge join \((\beta^L_{\alpha})\) operations and the global layer performs aggregating \((G)\) and sorting \((S)\) of the results.

In our Basic implementation using inverted index, we’ve adopted the same way of partitioning the whole dataset into subsets.

Example 5 (Contextual Index Partitioned on Entity Space): To answer the same query, the query will be issued on each local node. As the contextual index is still ordered by docid, 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 \(C_{\alpha^{p}}\) and \(C_{\alpha^{p}}\). Unlike in the document partition based approach, these matching can already be grouped and aggregated on the local nodes. This is because the partition scheme on the entity space naturally allows the grouping of entity results.
possible on the local nodes. And once the matchings are grouped, their local scores can be aggregated to form the query score for each distinctive entity tuple. In this example, the final query score of phone instance $p_8$, is calculated on node $1$ and that of $p_9$, is calculated on node $9$. Entity instances, along with their final scores, are send to the central processing unit, where the ranking is performed. As we can see, this design allows the possibility to move most of the computation on the central node down to the distributed local nodes.

As we have hinted in Section 3.2, this entity space partition scheme is based suited for entity search queries containing single entity type, which is the most basic and common entity query type. Now we discuss how to process query having multiple queries upon our contextual index partitioned on entity space scheme.

Let’s introduce an additional entity type #email to $Q_1$, which gives us the following query $Q_1'$: “ow20(amazon service #phone #email)”. Figure 11 shows the overall framework for processing $Q_1'$ upon the contextual index partitioned on entity space.

![Figure 11: Architecture for Processing Multiple-Entity Queries](image)

**Figure 11: Architecture for Processing Multiple-Entity Queries**

**Example 6 (Multiple-Entity Query Processing):** To process query $Q_1'$, the local nodes will perform as much processing as possible. In this case, the local nodes join the contextual sublists to produce “composite” contextual sublists (from multiple keywords to one entity type). Take local node 1 as an example, it will load contextual sublists $C_{\text{#phone}}$, $C_{\text{#email}}$, $C_{\text{#phone} \#\text{email}}$, and $C_{\text{#email} \#\text{phone}}$. It will then join lists $C_{\text{#phone}}$ and $C_{\text{#email}}$ to generate composite sublist $C_{\text{#phone} \#\text{email}}$. Similarly, composite list $C_{\text{#email} \#\text{phone}}$ will also be notice. This operation will again significantly reduce the list length. All the composite sublists will be sent up to the global layer. The composite sublists (e.g., $C_{\text{#phone} \#\text{email}}$) will first be merge to generate the overall composite list (e.g., $C_{\text{#phone} \#\text{email}}$). Then these composite lists (i.e., $C_{\text{#phone} \#\text{email}}$ and $C_{\text{#email} \#\text{phone}}$) will be joined as the same ways as joining contextual index. Results are then aggregated and sorted for final output. The final result is a tuple $(p_8, e_{81})$ with score 1.

Although this architecture for processing multiple-entity queries is more sophisticated and incurs more overhead than that for processing single-entity queries, it still significantly outperforms the Basic implementation, as we will show next in section 5.

5. **EXPERIMENTS**

In order to empirically study the effectiveness of our novel index design, contextual index, and parallel query processing framework for supporting efficient entity search, we have build a large-scale, distributed system using the YellowPage scenario on a real Web corpus. In this section, we will first briefly discuss the setup of our system, then, we will use three benchmark query sets to show that by leveraging contextual indexing and joining, we can speed up query processing by orders of magnitude.

5.1 **Experiment Setup**

To empirically verify that indexing and query processing design is effective for supporting efficient 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 49874 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 dual 500 Mhz Pentium III CPU, 1 GB memory and 160 GB of disk. 33 out of the 34 nodes are used as local indexing and processing units, whereas 1 node is used as the central aggregation node.

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 8,800,432 distinctive phone entity instances and 4,646,009 distinctive email entity instances.

We implemented both the Basic inverted index based approach and the contextual index based approach. In the inverted index based approach, we evenly distribute the whole corpus across the 33 local nodes by partitioning based on document IDs. In the contextual index based approach, we partition the contextual index on entity IDs. Each local node will be responsible for the same number of distinct entity instances for each entity type. For instance, email entity instances with ID in the range of 1 - 140,778 and phone entity instances with ID in the range of 1 - 266,680 will be indexed and only indexed on the first local node.

5.2 **Experiment Results**

To study the performance of our method in a systematic way, we use the following three benchmark query sets for evaluation. **Benchmark I (phone related):** 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. **Benchmark II (email related):** We use the names of 88 PC members of SIGMOD 2007 as part of our query, together with email entity type in the query. **Benchmark III (email & phone related):** We use the names of 88 PC members of SIGMOD 2007 as part of our query, together with email entity type and phone entity type in the query.

The reason why we select those three benchmark query sets are the following: First, those three benchmark query sets contains both simple single-entity queries (Benchmark I&II) and complex multiple-entity queries (Benchmark III). Second, the selectivity of the keywords also differs quite a lot. Keywords used in Benchmark I query set (e.g., “Walmart”, “Chevron”, etc) are far less selective than the ones used in benchmark I&II query sets (e.g., “Ailamaki”, “Chakrabarti”, etc). Third, the number of keywords in those three query sets also has good variation, in the range from one keyword to three keywords. Finally, all the queries used in those benchmark query sets are real queries and useful in practise. Overall, we believe those three benchmark query sets are typical and representative for a wide range of entity search queries.

To measure query processing on local processing units, where most of the computation are done, we look at the following four aspects for processing each query: List length - the size of index lists that is needed to load for processing query; Index loading time - the time needed to load index into disk; Joining time - the time needed to join the loaded index lists for producing entity tuples; Processing time - the time needed to load and join index for query processing. For each aspect, we measure the overall statistics summed over all the local processing units and the max statistic of all the local processing units. Take the processing time as an example, we measure both the sum of the processing time spent on all local processing units as well as the max processing time among all the local pro-
cessing units. The overall processing time indicates the throughput of the system, whereas the max processing time indicates the query response time of the system. Throughput and query response time are all primary measures for measuring search engines.

We define the selectivity of a query as the overall list length reduction using contextual index versus inverted index. The higher the reduction, the higher the selectivity.

To get robust experimental results, we execute each query 10 times. To eliminate the effect of cold start, the results from the first two runs are discarded. The results from the remaining eight runs are averaged and used for experimental study.

As we can see, there are strong correlations between the query selectivity and index loading time, joining time and processing time respectively. Figures 13 and 14 show similar result patterns for Benchmark II&III.

As we can see, most of the queries can be speedup by at least two orders of magnitude. We do observe less speedup in Benchmark II&III than Benchmark I. We believe this is mainly due to the difference in selectivity as the keywords used in Benchmark I are far less selective. Consequently, the saving in reducing list length as well as joining is not as much. We also observe the loading and processing speedup clearly flattens for queries that have high reduction in list length (where the contextual index are normally very short). This is because index loading consisting of random seek time and sequential read time. For queries that have short contextual lists, the random seek time plays a significant role.

In Figure 15(a), we report the query processing speedup with regard to the number of keywords in a query. To keep the number of entities a constant across all queries, we use all queries from Benchmark I and II. The number of keywords varies from one to three in the queries in Benchmark I and II. As we can see in Figure 15(a), all three query classes has speedup for more than two orders of magnitude on average regardless of the number of keywords. In Figure 15(b), we report the average query processing time comparison for all queries in Benchmark I, II and III. The queries are ordered in ascending order according to their average processing time using the contextual index approach. As we can see, contextual index based approach achieves two orders of magnitude over

Figure 12: Benchmark I (Phone Related)
Experiment results on query efficiency for the three benchmark query sets are shown in Figures 12, 13, 14 respectively. Now let’s zoom into Figure 12 and reveal details of the results.

Figure 12(a) shows the index list length reduction in log scale for each query. The queries are ordered in ascending order according to their selectivity. The queries in Figures 12(b), 12(c) and 12(d) are ordered according to their order in Figure 12(a). Figure 12(b) shows the index loading speedup in log scale for each query. Figure 12(c) shows the index joining speedup in log scale for each query. Figure 12(d) shows the query processing speedup in log scale for each query. As we can see, there are strong correlations between the query selectivity and index loading time, joining time and processing time respectively. Figures 13 and 14 show similar result patterns for Benchmark II&III.
In addition to experiment the local query processing speedup, we’ve also tested the network transfer cost between the local layer and the global layer, as well as the processing cost that occurs on the global layer. While we observe the contextual based approach generally requires much less network transfer cost as well as global processing cost, the costs are insignificant comparing to the local query processing costs and are therefore omitted for detailed discussion.

In terms of index size, our contextual index only is roughly 1/6 of the size of the inverted index. However, we need to point out that the index size is highly related with the number of entity types supported in the system. While in this YellowPage scenario, email and phone entity type are the only interesting entity types we consider, other application could support more entities types. In such cases, there is a chance that the contextual index size will overtake the inverted index size. However, we believe contextual index is still worth to utilize as nowadays disks are becoming cheaper while query processing is always in need for optimization.

6. RELATED WORK

We formulate the problem of entity search and emphasize its application on information integration in [6]. Our prototype search system is revealed in [8]. We study one of the core challenges for supporting entity search - the entity ranking problem - in [7]. We are now witnessing an emerging research trend on using entities and relationships to facilitate various search and mining tasks [4, 5, 17, 15, 13, 12, 1, 2, 3, 14, 19]. We have discussed the relationship between our work and these works in detail in the related work section of [7].

Our work is most related with the works on indexing unstructured documents. Inverted index [21] 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 [9] 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. Again, it is not optimized for phrase or proximity queries and still require full scan of candidate documents. Nextword index [18] is a structure designed to speed up phrase queries and to enable some amount of phrase browsing. It is an inverted index where each term list contains a list of the successor words found in the corpus. Each successor word is followed by position information. This index is optimized for answering keyword phrase queries. It doesn’t consider more flexible proximity based queries and doesn’t consider types other than keywords.

BE [1] develops a search engine based on linguistic phrase patterns and utilizes a specific “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. [5] introduce a class of text proximity queries and study scoring function and index structure optimization for such queries. Their study on index is more on reducing the redundancy, rather than improving efficiency. A recent work [10] 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 contextual index.

7. CONCLUSIONS

In this paper, we develop query indexing and processing for making entity search efficient and scalable. From the functional definition of entity search, we derive the operational definition, from both of which we derive our proposal. Unlike the natural baseline of indexing entities as keywords, we develop contextual joins, which materializes pre-joins between entities and keywords. We further develop data parallelization by entity-space partitioning, unlike the traditional document-space partitioning approach. Our experiments show significant 200-500 times of speedup and sub-second response time.

8. REFERENCES