Beyond Pages: Supporting Efficient, Scalable Entity Search with Dual-Inversion Index

Tao Cheng
tcheng3@cs.uiuc.edu

Kevin Chen-Chuan Chang
kcchang@cs.uiuc.edu

Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, IL 61801-2302

ABSTRACT

Entity search, a significant departure from page-based retrieval, finds data, *i.e.*, *entities*, embedded in documents directly and holistically across the whole collection. This paper aims at distilling and abstracting the essential computation requirements of entity search. From the dual views of reasoning–*entity as input* and *entity as output*, we propose a dual-inversion framework, with two indexing and partition schemes, towards efficient and scalable query processing. We systematically evaluate our framework using a prototype over a 3TB real Web corpus with 150M pages and over 20 entity types extracted. Our experiments in two concrete application settings show our techniques of on average, 2 to 4 orders of magnitude speed-up, over the keyword-based baseline, with reasonable space overhead.

1. INTRODUCTION

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

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

Web-based Question Answering (WQA) Question answering has moved much towards Web-based: Many recent efforts (*e.g.*, [3, 15, 27]) exploited the diversity of the Web to find answers for ad-hoc questions, and leverage the abundance to find answers by simple statistical measures (instead of complex language analysis). As requirements, to answer a question (*e.g.*, "where is the Louvre Museum located?"), WQA needs to find information of certain type (a location) near some keywords ("louvre museum"), and examine as many evidences (say, counting mentions) to determine the final answers.

Web-based Information Extraction (WIE) Information extraction, with the aim to identify information systematically, has also naturally turned to Web-based, for harvesting the numerous "facts" online—*e.g.*, to find *all* the ⟨museum, location⟩ pairs (say, ⟨Louvre, Paris⟩). Similar to WQA, Web-based IE exploits the abundance for its extraction: correct tuples will appear in certain patterns more often than others, as testified by the effectiveness of several recent WIE efforts (*e.g.*, [11, 4, 18]). As requirements, WIE thus needs to match text with contextual patterns (*e.g.*, order and proximity of terms) and aggregate matching across many pages.

Type-Annotated Search (TAS). As the Web hosts all sorts of data, as motivated earlier, several efforts (*e.g.*, [8, 4, 9]) proposed to target search at specific *type* of information, such as *person* names near "invent" and "television." As requirements, such TAS, with varying degrees of sophistication, generally needs to match some proximity patterns between keywords and typed terms and to combine individual matchings into an overall ranking.

We believe these emerging trends all consistently call for, as their requirements agree, a non-traditional form of search—which we refer to as *entity search* [9]. Such search targets at various typed unit of information, or *entities*, unlike conventional search finding only pages. In this paper, we use #-prefixed terms to refer to entities of a certain type, *e.g.*, #location or #person. Observing from WQA, WIE, and TAS, we note the unanimous requirements following the change of targets from pages to typed entities:

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

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

many times as it appears in numerous pages, which requires us to globally aggregate overall scores.

While the requirements for entity search have emerged, we have not tackled the computational challenges for efficiently processing such queries. Recent works have focused on effective scoring models mostly (*e.g.*, [8, 9]). For query processing, a widely adopted form (as in many WQA works [3, 15, 27]) is to "build upon" page search—to first find matching pages by keywords, and then scan each page for matching entities. This "baseline" (Sec. 3), much like *sequential scan*, is hard to scale, and thus it may work only by limiting to top-k pages— which will impair ranking effectiveness (Sec. 2).

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

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

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

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

2. ABSTRACTION & CHALLENGES

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

Functional Abstraction. An entity search system provides search over a set of supported entity types $\{E_1, \ldots, E_n\}$, which we informally consider as the *schema*. *E.g.*, our CSAcademia application in Sec. 6 has schema (#university, #professor, ...). Each type E_i is a set of *entity instances* that are pre-extracted from the corpus (*e.g.*, "201-7575" $\in$ #phone). As the requirements indicate (Sec. 1), we abstract entity search as follows:

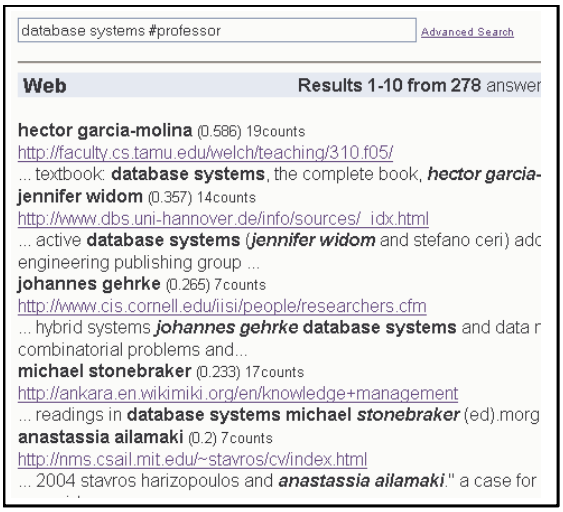


Figure 1: Result: "database systems #prof".

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

To illustrate, from our prototype (Sec. 6), Fig. 1 shows the screenshot for query Q_{db} "database systems #professor" (with default α as "order, 20-word window" written as ow20), for the first 5 results and supporting pages (where each answer appears). Notice, typically top results are supported by more than 1 page, as the ranking relies on aggregation. Fig. 1 shows one support page for each result just for conciseness.

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

- Entity as first class citizen: Unlike PS assuming page as *the* entity, ES can support any recognizable entity.
- Set as output: Unlike PS targeting at only a few *top* relevant results, an ES query can generally require a *set* of answers; *e.g.*, the above example (Fig. 1) can return tens or hundreds of relevant professors.
- Holistic aggregate: Unlike PS assuming each page as "unique," ES must generally handle entities occurring multiple times. Finding and returning such supporting evidences is crucial for applications, such as WQA, to actually determine the correct answers.

Computational Requirements. The objective of ES, as just abstracted, is to rank entity e (as instance of the target type E) by a scoring function $score(e)$. The choice of scoring function will directly impact the quality (or "relevance") of the ranked results. However, as this paper focuses on the computation framework, we will identify the key components of such ranking functions (Sec. 3 will give example scoring functions). Our previous work [9] addresses the quality of search results.

As the requirements of ES, as Sec. 1 identifies, a reasonable scoring function should capture both *context matching* and *global aggregation*. Consider scoring an entity e. For our discussion, let $o\langle doc, pos\rangle$ denote an *occurrence* of e in some page doc at word position pos (recall that an entity instance can occur many times in corpus D). Similarly, we use $\kappa_j\langle doc, pos\rangle$ as an occurrence of keyword k_j.

1. *Context matching:* The first step in scoring is to match the occurrences of k_i and e to the desired context pattern α. We assume a *local matching function* L_α. Given occurrences κ_i for keywords k_i and o for entity e, L_α will assess how well the positions match α by some similarity function $\text{sim}(\cdot)$, if the occurrences are in the same page.

$$L_\alpha(\kappa_1, \ldots, \kappa_m, o) = \begin{cases} 0, \text{if } \kappa_i.doc \text{ and } o.doc \text{ differ;} \\ \text{sim}(\alpha, \kappa_1.pos, \ldots, o.pos), \text{else.} \end{cases} \quad (1)$$

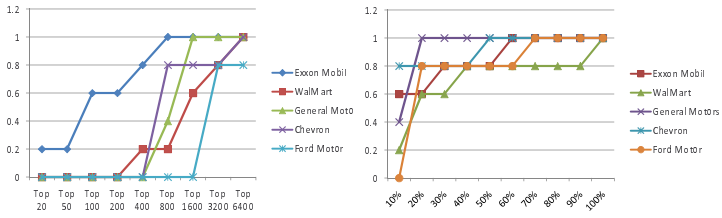
2. *Global aggregation:* The second step is to aggregate all the occurrences across pages. Here some function G aggregates the local scores globally into the total score, across all occurrences o in D.

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

$$score(e) = G_{(\kappa_1,\ldots,\kappa_m,o)\in d, d\in D}[L_\alpha(\kappa_1, \ldots, \kappa_m, o)], \quad (2)$$

Challenges. Document search has often relied on a small number of high quality documents for pruning, and therefore avoiding the need to scan full inverted lists (*e.g.*, [16]) for high efficiency. Such pruning techniques, however, are not directly applicable to entity search, with the mandate on processing comprehensive corpus due to the following two major reasons:

First, since entity search relies on *global aggregation*, comprehensive corpus is needed to generate stable aggregative statistics. Second, many entity queries are naturally looking for *set output* of comprehensive results over the entire corpus (*e.g.*, "#professor in DB" as in Q_{db} or "#city in California"). Figure 2(a) and 2(b) show the accuracy (measuring top 5 results) of 5 typical queries of finding the phone number of companies (represented by the y axis), by varying the number and percentage of top documents (returned by issuing keyword queries against a document search engine) used respectively (represented by the x axis). Evidently, different queries converge to accuracy 100% at very *different* points, indicating that it is nontrivial to determine which "top k" value to stop for different queries. Moreover, queries generally require a significant portion (over 40%) of all the relevant documents for stable results.



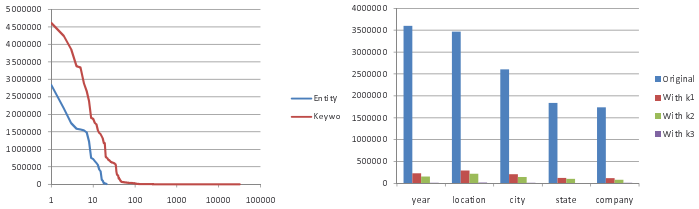
(a) Absolute Page Count (b) Page Percentage Count

Figure 2: Top K Comparison for Point Queries

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

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

Complex Join: As we see from the problem abstraction of entity search, each query involves at least one entity. Unlike keywords, entities, comprised of many entity instances, tend to appear frequently across the entire corpus. Figure 3(a) shows the comparison of keyword frequency (*i.e.*, the number of times a keyword appears in corpus) with entity frequency (*i.e.*, the number of times an entity type, say #phone, appears in corpus), with x axis in log scale representing keywords/entities under comparison, and y axis representing their respective frequencies. As seen from the figure, entities clearly appear much more frequently (by orders of magnitude) than most of the keywords, with frequency comparable to the top 20 most frequent keywords. Therefore, it is computationally expensive to load/check those many occurrences of entities for pattern matching. In addition, as discussed in the characteristics of entity search, it has to rely on in-document *contextual pattern matching*. Such computation is also more expensive, compared to the traditional simple document intersection checking in document search.



(a) Frequency Comparison (b) Selectivity of Keywords

Figure 3: Keyword and Entity

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

Global Aggregation: As we discussed in the characteristics in Sec. 1, entity search has to rely on *holistic aggregation* over comprehensive corpus to tap the rich redundancy of the Web. This is an online processing layer that is non-existent in traditional document search. Being able to support aggregating information over large-scale corpus in an online fashion is another essential computation requirement for entity search. How can we parallelize such online large-scale aggregation for scaling up?

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

3. BASELINE & RUNNING EXAMPLE

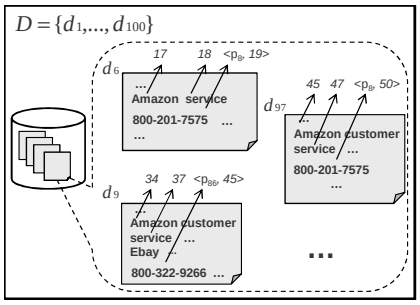


Figure 4: A running example: *YellowPage*.

To set the stage of discussion, let's use Fig. 4 as a running example throughout the paper, which we call the *YellowPage* scenario, as it provides search for contact information (*e.g.*, #phone, #email). As a toy dataset, the corpus D has 100 documents $D = \{d_1, \ldots, d_{100}\}$; we show three documents d_6, d_9, d_{97} as examples. We will assume a simple query for finding the phone number of Amazon service $Q1$ in the following form:

$Q1$: ow20 (amazon service #phone)

During offline processing, we recognize the position of keywords (*e.g.*, keyword "amazon" appears at position 17 of document d_6) in the corpus (via tokenization). Entities are also extracted offline, with their entity instances identified and positions recognized. *E.g.*, we extract phone number 800-201-7575 as phone instance p_8 at position 19 of document d_6 as shown in Fig. 4. Notice, there may be additional properties related with the extracted entity occurrences, *e.g.*, the extraction confidence. We exclude such information in this paper for the ease of discussion.

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

Example 1: [Scoring Function: *BinarySum*] Let's define scoring scheme *BinarySum*, which instantiates Eq. 2 by:

$$L_\alpha(e) = 1, \text{if } e \text{ matches } \alpha; 0 \text{ otherwise.}$$
$$G = Sum$$

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

To execute the general form of Eq. 2, as an entity-centric system is currently lacking, many related works (*e.g.*, [2, 3, 14, 15, 27]) have relied on keyword-based search engines to zoom into a subset of documents and then apply local matching by scanning documents and global analysis. Considering example $Q1$, the *baseline* goes as follows:

1. Look up by keywords (*e.g.*, "amazon service" for $Q1$) in a keyword-based search engine for retrieving documents matching these keywords. From the running example in Fig. 4, documents d_6, d_9 and d_{97} will be returned as matching documents.
2. Scan each matching document to execute local matching L_α to match candidate entities. Notice, as keywords and entities are all identified offline, only pattern matching (by pattern α) needs to be performed. In the example for instance, p_8 will be matched from document d_6 with local matching score of 1 by *BinarySum*.
3. Perform aggregation to assemble the produced matchings. In the example, p_8 will have an aggregate score of 2 from d_6 and d_{97} using *BinarySum*.

In order to handle large-scale dataset, it is common practice to partition the corpus into sub-corpses, and distribute the sub-corpses. Over the baseline, step 1&2 can be processed over the partitioned sub-corpse in parallel, while step 3 aggregates the results generated from the sub-corpses.

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

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

4. SOLUTIONS: DUAL-INVERSION INDEX

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

To begin with, we recognize that, for text retrieval, the key principle of indexing is *inversion*– an efficient data structure for mapping from query *input values* to *output objects*. In a standard text search scenario, users give *keywords* as input values and expect *documents* as output objects; *i.e.*, we are searching in a database of documents by keywords. Thus, the standard inversion that powers up today's text retrieval is mapping from keywords, as input values, to document as candidates for output objects. Since text databases are not optimized for real-time updates, an index does not need to be "dynamic," (unlike database indexes such as B-tree) and thus the most efficient data structure is simply a sequential list of such mappings, called *inverted list*—one list for each keyword—where each *posting* is one document ID and the positions in the document where k occurs. Such lists can be efficiently loaded from disk into memory by sequential read, or compressed and cached in memory [29].

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

$$D(k) : k \rightarrow \{\langle doc, pos\rangle | \ doc.content[pos] = k; d \in D\}. \quad (3)$$

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

4.1 Document-Inverted Index

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

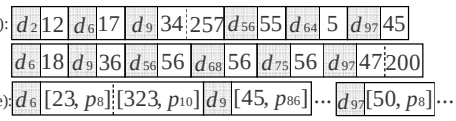


Figure 5: Document-Inverted Index Example

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

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

$$D(E) : E \rightarrow \{\langle doc, pos, entity\rangle |\ d.content[pos] = entity; \\ entity \in E; d \in D\}. \quad (4)$$

Fig. 5 shows the layout of the inverted lists $D(a)$, $D(s)$ and $D(\#p)$ for keywords "amazon", "service", and entity type "#phone" respectively.

For further development for query processing, we can conceptualize each inverted list as a *relation*: As Exp. 3 and 4 show, each list is simply a set of postings of the same structure—or "tuples"—and thus $D(k)$ is a relation with schema $\langle doc, pos\rangle$ and $D(E)$ a relation with schema $\langle doc, pos, entity\rangle$. Note that the relational view is conceptual, allowing us to understand the operations, and we do not necessarily use a DBMS to store and process the lists.

Computation Analysis. With this document-based inversion in place, we now capture the computation for query processing. Given the document-inverted lists $D(k_i)$ and $D(E)$, how do we process them to answer the query $\alpha(k_1, \ldots, k_m, E)$? We are starting from the D-Inverted lists as base relations $D(k_1), \ldots, D(k_m)$ and $D(E)$.

Specifically, we can now use relational operations to describe the essential operations. Starting from the base relations, our objective is to score every entity instance e by Eq. 2 and sort all the instances by their scores. First, to find all the qualifying entity occurrences, we perform join between the relations $D(k_1), \ldots, D(k_m)$ and $D(E)$. We call such join *context join* as it evaluates a context pattern α over the occurrences of $k_1, \ldots, k_m$, and some entity occurrence of E. It checks whether the occurrences match the pattern α and scores how well a matching is by the local scoring function L_α—thus, strictly speaking, it is a "fuzzy" join that returns scores. Second, we need a groupby operator $\mathcal{G}$ to group entity occurrences according to their instances, *i.e.*, $D(E).entity$, and use global aggregation function G to calculate the final score for each instance. Finally, a sort operator $\mathcal{S}_{score}$ sorts the entity instances. We show the overall computation in Exp. 5.

$$\mathcal{S}_{score}[_{(D(E).entity)}\mathcal{G}_G(\bowtie_{L_\alpha}[D(k_1), \ldots, D(k_m), D(E)])] \quad (5)$$

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

```
SELECT D(E).entity, G(mscore) AS score
FROM D(k1), ..., D(km), D(E)
WHERE Lα(D(k1).doc, D(k1).pos, ...,
  D(km).doc, D(km).pos, D(E).doc, D(E).pos) AS mscore
GROUP BY D(E).entity
ORDER BY score                                   (QES1)
```

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

```
SELECT R1.G1, ..., Rn.Gn, Agg(R1.A1, ..., Rn.An)
FROM R1, ..., Rn
WHERE Join(R1.J1, ..., Rn.Jn)
GROUP BY R1.G1, ..., Rn.Gn
HAVING/ORDER BY ...
```

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

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

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

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

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

Group-By (and aggregate) must happen after join, or we do not have all the aggregate attributes. While we explain intuitively, a full analysis of the feasibility of transformations is discussed in [28].

Data Partition: Document Space. To process entity search queries, now that we need to process aggregate after join, how to partition the relations for parallel query processing? To scale up entity search over a large corpus, we must partition data somehow over the p worker nodes. Our particular form of aggregate-join query is tricky for parallelization, because the join and group-by are over a different set of attributes—*i.e.*, in terms of the general form, $J_i \neq G_i$. To contrast, for Q_{bank}, since both the join and aggregate are over attribute *cid*, we can simply partition Customers and Accounts by the same *cid* ranges, and each worker node can execute both join and aggregate.

Unfortunately, when join and group-by are over different attributes, as in our situation, no schemes can fully partition the corpus for both join and aggregate without significant replication of communication. Naturally, we can partition on either join or aggregate attributes, as observed in [22, 23]. We next discuss these choices:

As the first choice, we may partition relations by their group-by attributes, which turned out to be infeasible for entity search. Referred to as *APM* [22], this aggregation partition method will partition each relation R_i by G_i. If R_i does not appear as part of Group-By (*i.e.*, $G_i = \emptyset$), then the entire R_i needs to be broadcast to all the nodes at run time (or otherwise every R_i needs to be replicated to every node). For entity search, as *offline data partitioning*, we partition $D(E)$ by $D(E).entity$ into sub-relations, $D^1(E)$, ..., $D^p(E)$, for the p local worker nodes; *i.e.*, records of the same entity instance will distribute to the same node. At *runtime processing*, for query $\alpha(k_1, \ldots, k_m, E)$:

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

$$_{(D^z(E).entity)}\mathcal{G}_G(\bowtie_{L\alpha}[D(k_1), \ldots, D(k_m), D^z(E)]) \qquad (6)$$

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

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

As the other choice, thus, we will partition by the join attributes. Referred to as *JPM* (join partition method) in [22], this method will partition each relation R_i by J_i. For entity search Q_{ES1}, we are matching pattern α by the context-join $\bowtie_{L_\alpha}$ over the keyword and entity relations on their *doc* and *pos* attributes. To determine the partition, we must examine— What are the conditions that these tuples from each relation are "joinable"—*i.e.*, $L_\alpha(D(k_1).doc, D(k_1).pos, \ldots, D(k_m).doc, D(k_m).pos, D(E).doc, D(E).pos) > 0$? Since we are matching entities and keywords from each document, any joinable occurrences must be at least from the same document. More formally, by the definition of L_α as Eq. 1 gives, the context join between $D(k_1), \ldots, D(k_m)$, $D(E)$ must require that

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

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

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

Each local node will host the corresponding sublist for each keyword, and entity. For instance, the document-inverted index of entity #phone in Fig. 5 will be split into p sublists, and the i-th sublist will be located on the i-th local node. For the *YellowPage* scenario, assuming we have 10 local processing units, we can partition the dataset containing 100 document into 10 subset, each containing 10 documents as shown in Fig. 6. This implies the inverted index will be partitioned into sublists. For instance, $D(a)$ in Fig. 5 will be partitioned into 10 sublists, $D^1(a)$, ..., $D^{10}(a)$ respectively.

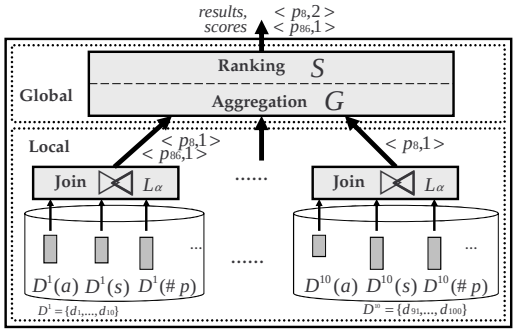


Figure 6: Partition by Document Space

Parallel Query Processing. The local processing module, having all the information of a subset of the documents, will be able to compute all the *context join*s and output all the matching entity occurrences. Exp. 7 formulates this procedure, where the matching entity occurrences are put into L^z and will be sent over to the global processing module for further processing. This step implements the context join operation in Exp. 5 in parallel across local nodes.

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

$$L^z\langle entity, mscore\rangle = \pi_{entity,mscore} \bowtie_{L_\alpha} \text{as } mscore [D^z(k_1), \ldots, D^z(k_m), D^z(E)] \qquad (7)$$

The local matching algorithm D-Local in Fig. 7 loads the document-inverted index for the specified keywords, and entity into memory (step 2). As the lists are sorted based on document id, merge-join can be performed over the lists to instantiate any possible matchings (step 3-10). If a matching entity occurrence is found, we will use local scoring function L to compute the score, and output (step 5-8).

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

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

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

Figure 7: Algorithm D-Local.

To answer query Q_1, we will execute the query on each of the local nodes as shown in Fig. 6. Local node 1 will produce two matchings for phone instance p_8 matched in document d_6 and p_{86} matched in document d_9 by joining sublists $D^1(a)$, $D^1(s)$ and $D^1(\#p)$. Local node 10 will produce one matching for phone instance p_8 matched in document d_{97}.

With the entity occurrences and local scores produced, we are ready to perform *holistic* aggregation over them. The global processing module takes care of both aggregation and sorting over all the matching entity occurrences in $L^1, \ldots, L^p$ collected from all the local nodes, as shown in Exp. 8:

Global Node: :

$$\mathcal{S}_{score}[_{(entity)}\mathcal{G}_{G(mscore)\ \text{as}\ score}(L^1 \cup \cdots \cup L^p)] \quad (8)$$

The global aggregation algorithm D-Global in Fig. 8 goes through all the input matched entity occurrences, and aggregates all the scores of a specific instance together. As shown in Fig. 6, the global processing layer receives the matching occurrences from the local nodes, and performs aggregation and ranking. For instance, the local scores for p_8 are aggregated into the final score of 2, resulting the ranking of p_8 at the first place.

4.2 Entity-Inverted Indexing

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

Indexing: Entity Inverted In the functional form, entity search finds entity instances as output from keywords as input: Given query $\alpha(k_1, \ldots, k_m, E)$, we are looking for entities e of type E, such that keywords k_i appear in the *context* of e in a way that matches pattern α. *E.g.*, in our example, we are given "amazon service" to search for entity #phone that are mentioned with these keywords around it (in that sequential pattern).

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

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

1: $Result = \emptyset$
2: **for** each $\langle entity, mscore\rangle$ in $L^1, \ldots, L^p$ **do**
3: **if** *entity* not in *Result* **then**
4: add *entity* to *Result*
5: **end if**
6: update $Result[entity].score$ with *mscore* by G
7: **end for**
8: sort *Result* by *score*; return *Result*

Figure 8: Algorithm D-Global.

With this view, we again seek to parallel the traditional inversion. We observe that traditional document search builds upon keyword inversion $D(k)$, as Exp. 3 shows, which maps each keyword k as query input to documents in D as output. For entity search, we shall map each keyword k to entities $\in E$, denoted $E(k)$. As the inversion targets to entities, we call $E(k)$ an *entity-inverted* index, or *E-Inverted* index for short.

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

$$E(k) : k \rightarrow \{\langle o\langle doc, epos, entity\rangle, pos\rangle | o.context[pos] = k; entity \in E\}. \quad (9)$$

As the second issue, we also must define what *context* means—*i.e.*, how far from an entity occurrence shall we consider as *within* its context? We note that, for our first document-based inversion, the *content* of a document is well defined. Here, to define the "context" of an entity occurrence o, we are essentially considering the question—How far apart between k and o do we consider them as no longer "semantically associated"? Clearly, larger the distance is, the less likely they are associated, and most entity-oriented search efforts (*e.g.*, [8, 9]) leverage this insight in ranking. Thus, in our indexing, we can choose some maximal window distance to consider as context. In our implementation, we use 200-word window as the context—*i.e.*, the context of an entity occurrence extends between 100 words to its left and 100 to its right. Fig. 9 shows the layout of the entity-inverted index using our example.

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

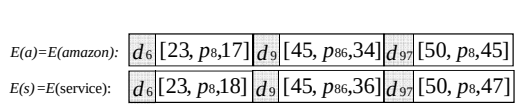


Figure 9: Entity-Inverted Index Example

$pos\rangle$.

Computation Analysis.

Starting from these base relations, in contrast to Exp. 5, we can express the computation of entity search for $\alpha(k_1, \ldots, k_m, E)$ as:

$$\mathcal{S}_{score}[_{(D(k_1).entity)}\mathcal{G}_G(\bowtie_{L_\alpha}[E(k_1), \ldots, E(k_m)])] \qquad (10)$$

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

```
SELECT E(k1).entity, G(mscore) AS score
FROM E(k1), ..., E(km)
WHERE Lα(E(k1).doc, E(k1).epos, E(k1).entity, E(k1).pos, ...,
        E(km).doc, E(km).epos, E(km).entity, E(km).pos)
        AS mscore
GROUP BY E(k1).entity
ORDER BY score                                   (QES2)
```

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

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

Data Partition: Entity Space.

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

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

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

Parallel Query Processing. Upon the entity space partition scheme, the local processing module can perform the joining operation, as well as the aggregation operation. In other words, Exp. 10 can be fully realized at each local node (except for the final ranking part):

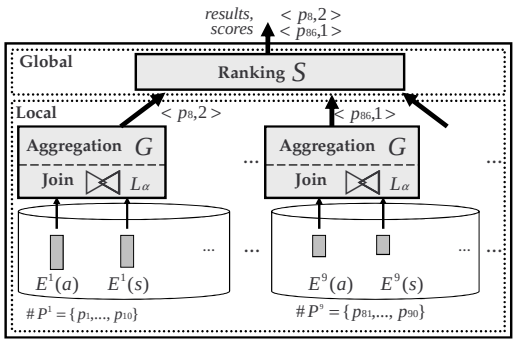


Figure 10: Partition by Entity Space

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

$L^z\langle entity, score\rangle =$

$$\pi_{entity,score\ (E^z(k_1).entity)}\mathcal{G}_{G(\text{mscore})\ \text{as}\ score}(\bowtie_{L_\alpha\ \text{as}\ \text{mscore}}[E^z(k_1), \ldots, E^z(k_m)]) \qquad (11)$$

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

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

```
1:  L^z = ∅
2:  load lists E^z(k1), ..., E^z(km)
3:  for merge-join over the loaded lists do
4:    if E^z(k1).o = ... = E^z(km).o then
5:      let o be the entity occurrence in common
6:      if E^z(k1).pos, ..., E^z(km).pos, o.epos match α then
7:        mscore = Lα (E^z(k1).pos, ..., E^z(km).pos, o.epos)
8:        if o.entity not in L^z then
9:          add o.entity to L^z; initialize score to 0
10:       end if
11:       update entity's score with mscore by G
12:     end if
13:   end if
14: end for
15: return L^z
```

Figure 11: Algorithm E-Local.

We illustrate the local matching&aggregation algorithm in Algorithm E-Local in Fig. 11. It loads the entity-inverted index for the specified keywords with regard to the input entity (step 2). As the lists are sorted based on document id, merge-join can be performed over the lists to instantiate any possible matchings (step 3-14). If a matching entity occurrence is found, we will use local scoring function L to compute the score (step 6-7). This score will be immediately aggregated with the produced occurrences (step 8-11).

To answer the same query, the query will be issued on each local node as shown in Fig. 10. As the entity-inverted index is still ordered by document id, the same sort-merge join algorithm can be applied. In this setting, the two matchings of phone instance p_8 will both be produced from local node 1 by joining sublists $E^1(a)$ and $E^1(s)$. Unlike in the document partition based approach, these matching can already be grouped and aggregated on the local nodes. In this example, the final query score of phone instance p_8 is calculated on node 1 and that of p_{86} is calculated on node 9.

Given that the local processing module produces aggre-

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

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

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

Figure 12: Algorithm E-Global.

gated results, the global processing module only has to take care of the ranking step in Exp. 10 of all the aggregated results from $L^1, \ldots, L^p$, a very light-weight task as shown in Exp. 12 and algorithm E-Global in Fig. 12:

$$\underline{\textbf{Global Node:}}: \quad \mathcal{S}_{score}[L^1 \cup \cdots \cup L^p] \tag{12}$$

4.3 Together: Dual-Inversion Index

We summarize the pros and cons of D-Inverted and E-Inverted proposals in terms of the computation requirements we listed in Sec. 2, pattern join, aggregation, as well as the space requirement, in the following table:

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

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

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

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

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

5. RELATED WORK

We are now witnessing an emerging research trend on using entities and relationships to facilitate various search and mining tasks [7, 8, 25, 13, 12, 4, 5, 6, 20, 27, 9, 30].

Our work is most related with the works on indexing unstructured documents. Cho [10] builds a multigram index over a corpus to support fast regular expression matching. A multigram index is essentially building a posting list for selective multigrams. It can help to narrow down the matching scope. It is not optimized for phrase or proximity queries and still require full scan of candidate documents. Nextword index [26] is a structure designed to speed up phrase queries and to enable some amount of phrase browsing. It does not consider more flexible proximity based queries and does not consider types other than keywords. Indexing keyword pairs to speed up document search is studied in [17]. Our motivation to speed up entity search is different from their goal and therefore the frameworks also differ. Our index design considers entities beyond keywords, where we introduce the unique entity space partition scheme. BE [4] develops a search engine based on linguistic phrase patterns and utilizes a special "neighborhood index" for efficient processing. Although BE considers indexing types such as noun phrases other than keywords, its index is limited to phrase queries only. Chakrabarti et al. [8] introduce a class of text proximity queries and study scoring function and index structure optimization for such queries. Their study on index design is more on reducing the redundancy and the index is used for performing local proximity analysis without considering global aggregation and multi-node parallelization. Comparing with our own work [30] on supporting content querying with the design of content query language (CQL), this work focuses on the principles and foundation for the index design for facilitating efficient entity search. Moreover, this work also studies distributive computation with parallelization schemes.

There are many existing optimization techniques in IR, such as caching ([19, 17]), pruning ([21, 16]), *etc.*, to improve the efficiency of document search. Such techniques are either orthogonal to our problem, *e.g.*, caching, or can not be directly applied in our setting which requires processing over comprehensive corpus as we discussed in Sec. 2, *e.g.*, pruning. It is the unique computation requirements of entity search, which distinguish it from document search, that motivate us to develop novel solutions.

Since our entity search query can be viewed as "aggregate-join query" from the DB perspective, our work is also naturally related with DB literature on handling such queries ([28, 22, 24]). Such techniques are mainly designed for a small number of relations under DB setting. Our work innovates upon these works in a rather different setting: an IR setting of inverted indexes where there are almost uncountable number of keywords.

6. EXPERIMENTS

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

chines, each with a dual AMD Athlon 64 X2 3600+ CPU, 1 GB memory and 1TB of disk.

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

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

Table 1: Supported entity types: 21 entities.

For our comparison, we implemented all the three approaches discussed: the keyword-based Baseline (Sec. 3) and the dual-inversion: D-Inverted and E-Inverted index (Sec. 4). As Table 2 summarizes, all the three methods, including Baseline, had the entities pre-extracted offline. As indexes, the Baseline used standard keyword inverted lists $D(k)$, and D-Inverted added $D(E)$ in addition, while E-Inverted used only keyword-to-entity inversion $E(k)$. (We will compare the space requirements later.) All the methods are parallelized across the same $(p+1)$-node cluster, by partitioning the index data as we discussed.

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

Table 2: Indexes built for each method.

Experiment Setup. To extensively and realistically study the performance, we configured two concrete applications. We evaluated 4 benchmark sets, for totally 176 queries of varying parameters. Each query has the form $\alpha(k_1, \ldots, k_m, E)$, as Sec. 2 defines, for keywords k_i and entity type E. We use "ow20" for pattern α— ordered 20-word window— for all queries. As scoring function, we use the "EntityRank" model [9], which is of the common form of $G \circ L_\alpha$ as Sec. 2 defines. We stress that the actual function affects "only" ranking preciseness. For our focus of efficiency, all functions with the join-then-aggregate (L_α then G) abstraction are computationally similar.

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

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

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

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

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

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

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

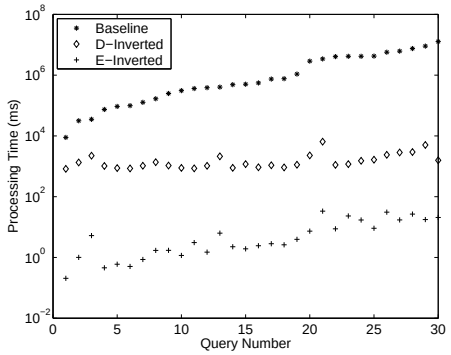
While we focus on efficiency, we note that the usefulness of entity search is also revealing through these benchmarks. *E.g.*, As Sec. 2 mentioned, Fig. 1 shows the screenshot for "database systems" #professor with supporting pages. Such queries, with page search, would require us to comb through numerous page results to collect answers. Entity search expands our ability to directly find fine grained information holistically across many pages.

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

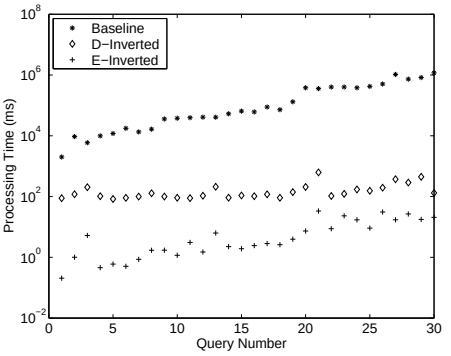
Fig. 13 shows the local times for 1A (queries are sorted by overall local processing time in Baseline). Both D-Inverted and E-Inverted incur much less overall and max local processing time than Baseline, and E-Inverted performs faster than the D-Inverted. As the graphs are in log scale, we observe rather significant speedup–generally *two* orders of magnitude: E-Inverted ranges around $10^2 ms$, D-Inverted $10^4 ms$, and Baseline $10^6 ms$. Furthermore, the times for D-Inverted and E-Inverted are more uniform than the Baseline, which has high variance in the number of documents needed to scan after keyword lookup.

Fig. 14 shows the transfer times for 1A. Notice, the cost for Baseline and D-Inverted are the same (thus the points collapsed together), since they send the same partial "after-join" results to the global node. We observe that E-Inverted can save significantly in network transfer cost, as results are already "after-aggregation." The difference is, again, significant–at about *two* orders of magnitude. Notice, in the case of only outputting top-k results, E-Inverted scheme can further save transfer cost, as at most top-k results from each local node need to be sent for final ranking.

Fig. 15 shows the global times for 1A. E-Inverted requires much less global processing time compared with the Baseline and D-Inverted (which have the same global costs). The difference is about *one* order of magnitude.

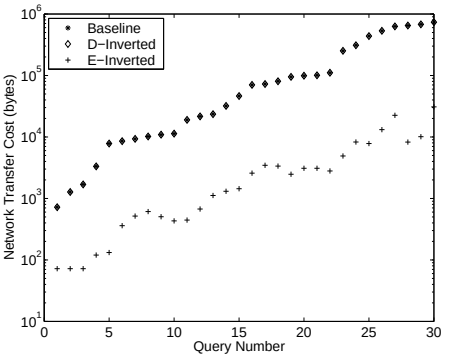


(a) M1: Overall Local Processing.

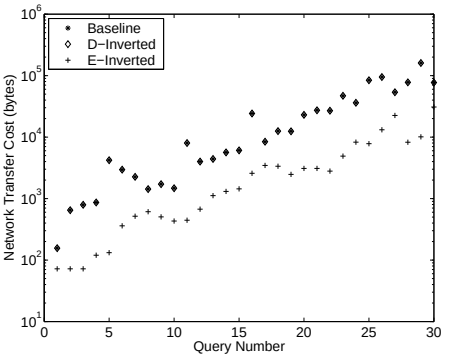


(b) M2: Max Local Processing.

Figure 13: Local Processing: Benchmark 1A.



(a) M3: Overall Network Transfer.



(b) M4: Max Network Transfer.

Figure 14: Network Transfer: Benchmark 1A.

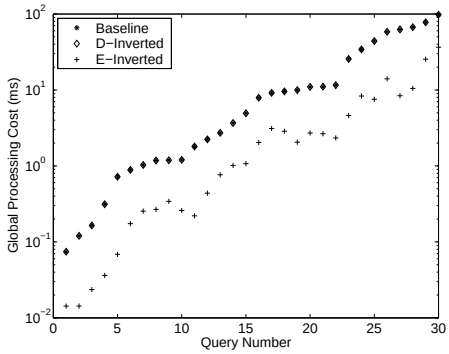


Figure 15: Global Processing (M5): Benchmark 1A.

Overall, we observe similar results for all the benchmarks, 1A, 1B, 2A, and 2B, in both applications. Table 3 summarizes the *median* cost of all the $M1$ to $M5$ metrics. We consistently observe, from Table 3, across the four benchmarks of totally 176 queries, the significant speedup of the dual-inversion approaches, for all the processing components $M1$ to $M5$. Both E-Inverted and D-Inverted are much faster than the Baseline– which use keyword indexes to look up pages for entity search.

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

Table 3: Summary of metrics.

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

The speedup comes at the cost of indexing entities—recall index configuration in Table 2. Table 4 also compares the various index sizes of the two application settings. *First*, we observe that, since D-Inverted relies on $D(E)$ in addition to standard keywords $D(k)$, it always requires larger index size than Baseline–However, the addition is actually

quite small, resulting in 1% and 0.1% size increase in Application 1 and 2, respectively. *Second*, we observe that, with its entity-primary indexing on $E(k)$, E-Inverted can require varying indexing sizes, depending on the actual entities indexed. In Application 1, E-Inverted requires 89.7% more space, while it actually save space in Application 2 with a reduction of 80.7% index size. The variation comes from varying *selectivity* of an entity: Some entities are very frequent, such as #location in Application 1, which result in long entity-inverted indexes. Other more "specialized" entities are much less frequent, such as #university in Application 2.

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

Table 4: Overall: Time and space.

Overall, the experiments conclude that both types of inversions can significantly speed up entity search, while keeping space overhead acceptable. The dual-inversions, D-Inverted and E-Inverted, also present interesting tradeoff: D-Inverted generally requires minimal space addition, while E-Inverted constantly achieve higher speedup. As Sec. 4 discussed, both types of inversion can coexist, to balance the tradeoff– *E.g.*, in a system supporting both Application 1 and 2, we may use D-Inverted for Application 1 and E-Inverted for Application 2, resulting in small space overhead and large speedup.

7. CONCLUSIONS

In this paper, we presented the dual-inversion framework, with two index structures document-inverted index and entity-inverted index, their respective data partitioning schemes and query processing. Extensive experiments show the techniques can support efficient and scalable entity search.

8. REFERENCES

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

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

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

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

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

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

[7] S. Chakrabarti. Breaking through the syntax barrier: Searching with entities and relations. In *ECML*, 2004.

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

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

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

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

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

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

[14] C. C. T. Kwok, O. Etzioni, and D. S. Weld. Scaling question answering to the web. In *WWW*, 2001.

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

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

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

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

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

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

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

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

[23] D. Taniar, R. B. Tan, C. H. C. Leung, and K. H. Liu. Performance analysis of "Groupby-After-Join" query processing in parallel database systems. *Information Sciences*, 168(1-4), Dec. 2004.

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

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

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

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

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

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

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