Ranking Objects Based on Relationships and Fixed Associations

Albert Angel
University of Toronto
albert@cs.toronto.edu

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Gautam Das
University of Texas at Arlington
gdas@cse.uta.edu

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

ABSTRACT

Text corpora are often enhanced by additional metadata which relate real-world entities, with each document in which such entities are discussed. Such relationships are typically obtained through widely available Information Extraction tools. At the same time, interesting known associations typically hold among these entities. For instance, a corpus might contain discussions on hotels, cities and airlines; fixed associations among these entities may include: airline A operates a flight to city C, hotel H is located in city C.

A plethora of applications necessitate the identification of associated entities, each best matching a given set of keywords. Consider the sample query: Find a holiday package in a "pet-friendly" hotel, located in a "historical" yet "lively" city, with travel operated by an "economical" and "safe" airline. These keywords are unlikely to occur in the textual description of entities themselves, (e.g., the actual hotel name or the city name or the airline name). Consequently to answer such queries, one needs to exploit both relationships between entities and documents (e.g., keyword "pet-friendly" occurs in a document that contains an entity specifying a hotel name H), and the known associations between entities (e.g., hotel H is located in city C).

In this work, we focus on the class of "entity package finder" queries outlined above. We demonstrate that existing techniques cannot be efficiently adapted to solve this problem, as the resulting algorithm relies on estimations with excessive runtime and/or storage overheads. We propose an efficient algorithm to process such queries, over large corpora. We devise early pruning and termination strategies, in the presence of joins and aggregations (executed on entities extracted from text), that do not depend on any estimates. Our analysis and experimental evaluation on real and synthetic data demonstrates the efficiency and scalability of our approach.

1. INTRODUCTION

In many application domains, such as e-commerce, social networking sites, digital libraries and collaborative knowledge repositories, to name a few, metadata relate (unstructured) textual documents to the real-world *entities* discussed in them. For instance, in Wikipedia, the well-known collaborative encyclopædia, the unstructured document of an article about a person is related to entities such as the person, a birthplace, past employment institutions etc. In other domains, such as news articles, blog posts, etc. such document-entity relationships can be obtained through widely available Information Extraction tools ([4], [1]), which automatically identify named entities discussed in a document (e.g. Person, City, Company, Product).

Such entities are related via *fixed associations* typically known among them. For instance, a table in a relational database can associate neighbourhoods, houses for sale, and schools, via their location, thus giving rise to *packages* of associated entities (in this case, home-neighbourhood-school packages). Such information can be retrieved from several sources, e.g. from corporate databases, a collaborative public knowledge repository such as Freebase [8], etc. Moreover, it can be either *static* (e.g. school A is located in neighbourhood B) or *dynamic* (e.g. flights X and hotel Y are currently offered as a discounted holiday package).

In several instances, the goal is to identify packages whose entities each best match a given set of keywords. For instance, using comments from real-estate listings, blogs, etc., one may wish to identify a "Victorian", "3 bedroom" house for sale, located in a "low-crime", "safe" neighbourhood, that has a school with "bilingual education" and a renown "swimming team". As another example, using reviews from a travel planning site, one might wish to identify a holiday package, consisting of a destination that is generally considered to be "historical" and "lively", a hotel that is "pet-friendly", and travel operated by an airline renown for its "economical" and "safe" flights. Let us examine this motivating example of finding personalized holiday packages in greater detail.

Consider a corpus of reviews taken from a travel-planning site, shown in fig. 1. We assume that entities, such as hotels, cities and airlines, have been automatically identified in individual reviews, using a Named Entity Extraction tool. Moreover, we assume a relation containing known associations between entities, e.g. that hotel H is located in city C, or that airline A flies to city C. This relation can be available as a database table in the travel-planning site's infrastructure, or it can be retrieved from some external source (e.g. [8]). Furthermore, some parts of it might change very rarely, if at all (e.g. hotel H is located in city C), whereas others can be subject to frequent updates (e.g. a discount holiday package containing a flight by airline A, and a stay at hotel H, is currently being offered). Our database thus consists of *documents*, representing reviews, and *entities* of three *types*, Hotels, Cities and Airlines. *Relationships* between documents and entities are represented by three tables of

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

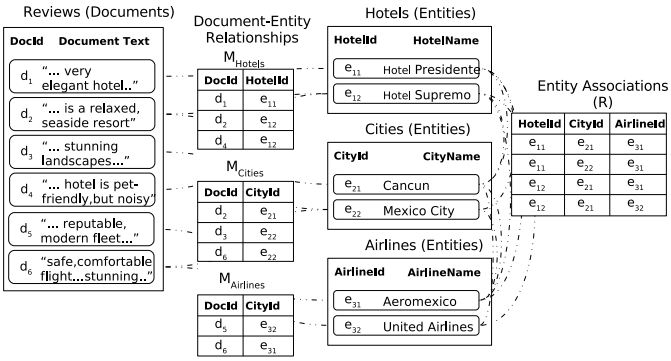


Figure 1: Data model

$(DocId, EntId)$ tuples, one table for each entity type. (For instance, a tuple (d, e) in table Hotels signifies that the review with id d discusses the hotel with id e - i.e. e has been extracted as a topic of discourse from the text of d). Moreover, known *associations* between entities can be represented in a variety of ways, e.g. three tables denoting pairwise associations, etc. We abstract their representation by assuming a view R over them. R contains (e_1, e_2, e_3) *package tuples*, signifying that hotel e_1, city e_2 and airline e_3 are associated, i.e. hotel e_1 is located in city e_2, airline e_3 flies to city e_2, and a flight with airline e_3 can be combined in a holiday package with a stay at hotel e_1. In fig. 1, a user might want to poll the public opinions, expressed in reviews, to find a holiday package in a city that is "historical", yet "lively", with a stay at a hotel that is "pet-friendly", and with travel operated by an "economical" and "safe" airline. These descriptive terms are unlikely to occur within the names of cities, hotels, or airlines; thus standard keyword search techniques cannot be used to answer such queries. Moreover, a user is not likely to be interested in all possible answers to their queries, but would prefer to be presented only with a small number of the most relevant results. We term this type of queries as *Entity Package Finder* queries (EPF). The necessary data to support such queries is a collection of:

1. *Documents* , which are searched by keywords;
2. *Entities*, each having a Type attribute (e.g. in fig. 1 Hotel Presidente is an entity of type Hotel);
3. *Relationships* between documents and entities, expressed as document-entity pairs, and denoting that a document refers to an entity; they can be static or dynamic; and
4. *Associations* among entities of different types, expressed as a relation containing tuples of entities; we refer to such tuples as *packages*

The aim in entity package finder queries is to identify the k most relevant packages of entities, with respect to each entity best matching a given set of keywords. This gives rise to the need to score packages, entities and documents with respect to the given keyword query. The score of a document wrt. a keyword query can be assessed using standard Information Retrieval techniques, such as textual similarity [17]. Note that entities are typically not directly related to keywords; for instance, named entity extraction from a document is typically based on more complex features than the simple presence of certain keywords [4]. For this reason, we will say that an entity matches a keyword query if it is related to documents that match the query. The quality of this match will depend both on the number of documents matching the keywords, that are related to the entity, as well as on the strength of the match between each such document and the given keywords. Thus, the score of the entity will be an aggregate of the scores of all documents matching the given keywords that are related to it. For example, in fig. 1, the score of e_{12} (Hotel Supremo) wrt. keyword query "relaxed" will be an aggregate of the scores of documents d_2, d_4 wrt. the keyword "relaxed". Finally, to capture the notion that relevant packages are those that contain, overall, relevant entities, packages are scored based on an aggregate of their entities' scores (e.g. in fig. 1, the score of the holiday package (e_{11}, e_{21}, e_{31}) wrt. some query will be an aggregate of the scores of e_{11}, e_{21} and e_{31} wrt. the query). In this work we use a scoring framework that encompasses a wide class of scoring semantics.

The essence of the entity package finder problem is to find the top-k packages with the highest scores, that result from score *aggregations* over documents matching a given query, and subsequent *joins* between entities of different types, according to a set of fixed associations. The techniques we propose are applicable for general entity finder queries, on any domain where documents can be searched by keywords (e.g. textual corpora, multimedia databases, medical databases etc.). In the following, solely for clarity of presentation, we will restrict ourselves to the case of text corpora.

Whereas this problem can be solved using standard RDBMS technology, the resulting solution would be highly inefficient. This is due to the fact that, in such a scenario, the precise scores of every entity and package would need to be calculated, followed by a selection of the top-k packages. Instead, early termination and pruning techniques can be used, to drastically reduce the necessary processing effort.

Adapting existing such techniques to this problem also leads to inefficient algorithms, as they require estimations with impractical runtime and/or storage overhead (see sec 3). To overcome this difficulty, we utilize the following intuition: Top scoring packages are expected to receive most of their score from top scoring entities, which, in turn, are expected to receive most of their score from top scoring documents. In our approach, documents are processed in descending score order, with respect to the given keyword query. Using document-entity relationships and entity associations, the aggregate scores of entities and packages are computed, incrementally, and in a rank-aware fashion. Periodically *tight bounds* on these scores are computed, and used to prune entities and packages that cannot rank among the final top-k. Thus, our algorithm is able frequently to terminate after processing only a small fraction of the input.

Our main contributions in this paper can be summarized as follows:

- We introduce the Entity Package Finder type of queries, able to answer useful questions with intuitive semantics.
- We formalize the EPF problem in a threshold algorithm framework, and demonstrate that existing techniques cannot be efficiently adapted to solve this problem, as the resulting algorithm relies on estimations with impractical overhead (runtime, storage).
- We propose a complementary early pruning/stopping approach, which interleaves rank-join and aggregation, and overcomes the need for such estimates. Our approach exploits all available knowledge regarding possible entity packages to provide tight bounding on package scores, leading to increased pruning efficiency.
- We analytically and experimentally evaluate the performance

of our algorithm, on both real and synthetic data, and demonstrate its efficiency and scalability.

The rest of the paper is organized as follows: We formalize the EPF problem, and provide a threshold algorithm framework for it in section 2. We demonstrate the inapplicability of existing techniques (section 3), and propose an efficient algorithm in section 4. We discuss extensions to our algorithm and problem in section 4.2. The efficiency and scalability of our algorithm is demonstrated by analysis (section 5.1), as well as thorough experimental evaluation (section 5.2), on both real and synthetic data. We review other related work in section 6, and conclude in section 7.

2. A FRAMEWORK FOR EPF

Given the entity package finder class of queries introduced above, we subsequently formalize our problem, and describe a threshold algorithm framework for processing such queries.

2.1 Data model

Every *document* (e.g. review, blog post, etc.) in the corpus, is associated with a document id d. Every document contains *search terms* (keywords) (e.g. d_1 in fig.1 contains keywords "very", "elegant" and "hotel", among others), and every document-term pair is associated with a score. These scores capture the importance of the term in the document, and can be derived using standard IR measures [17]. We assume the availability of an *inverted index* over the document corpus, i.e., for every search term, there exists an efficient way to retrieve all documents containing it, in descending score order.

Additionally, real-world *entities* are represented in our corpus. Every entity has an entity id e, and belongs to one of a number of *types* $T_1, T_2, \cdots, T_n$. For instance in fig.1 Hotel Presidente is an entity of type Hotel. In addition, n relational tables M_{T_i} contain the *relationships* holding between documents and entities of type T_i. For example, in fig.1, table M_{Hotels} contains tuples of the form (d, e), signifying that document d mentions hotel e.

Finally, fixed *associations* between entities are available. Associations can be stored in a variety of ways (e.g. one or more relational tables), and can be either static, or dynamic (i.e. frequently updated). In either case, we assume the existence of a view, R over this data. R is an n-ary relation, with domain $T_1 \times \cdots \times T_n$, and describes the union of all valid query answers: every tuple $(e_1, \cdots, e_n) \in R$, termed a *package*, contains associated entities. E.g., in fig.1, entities e_{11}, e_{21} and e_{31} form a valid holiday package. As R might not be materialized in its entirety, we only require that it be accessible through a restricted API, able to efficiently return the packages containing a given entity e in a given type T.

In the following, we assume that all M_{T_i} tables, as well as the information necessary to provide access to R, fit into main memory, noting that our algorithms can be extended to handle cases where these assumptions do not hold[1].

[1]This is a reasonable assumption, even for large-scale corpora. A typical document discusses a small number of entities, and for every such relationship we only need to maintain a document id and an entity id. Thus, the total memory overhead of M_{T_i} tables will be a small multiple of the number of documents. This is consistent with our empirical observations on a large scale corpus of real data (see sec 5.2, $REAL$ corpus), where the memory overhead of M_{T_i} tables was, on average, under 10 bytes per document. Moreover, we expect the base information behind R to fit in main memory, due to the semantics of the entity package finder problem. This,too, is consistent with our empirical observations; the $REAL$ corpus, for instance, contained 110K associations, requiring under 1MB of main memory.

2.2 Query and Result Model

An entity package finder *query* is an n-ary tuple containing sets of search terms, i.e. a tuple of the form: $(W_1, W_2, \cdots, W_n)$, where every W_i is of the form $W_i = \{w_1, w_2, \cdots\}$. For example, in the holiday package case, a query could be ({"pet-friendly"}, {"historical", "lively"}, {"safe"}), meaning that the user would like to find a package with a "pet-friendly" hotel, a "historical" and "lively" city, and a "safe" airline.

An answer to such a query is a valid package, such that each of its entities, i, is related to at least one document d, containing some keyword in W_i. More formally, an answer is a tuple $(e_1, e_2, \cdots, e_n) \in R$ such that, for all $i \in \{1, 2, \cdots, n\}$, there exists a document d containing a term $w \in W_i$, such that $(d, e_i) \in M_{T_i}$. For instance, in fig. 1, (e_{11}, e_{22}, e_{31}) is a valid answer to the query ({"elegant"},{"stunning"},{"safe"}). Of course, the more keywords from W_i d contains, and the more such documents each entity e_i is related to, the higher the relevance of the answer to the query. Additionally, since users are typically interested only in seeing a small number, k, of the most relevant answers to a query, the need to score answers arises.

2.3 Scoring Answers

Answer scoring proceeds in two levels: Firstly, entities of the same type need to be scored, based on the degree to which the documents that are related to them, match the given search terms. Thereafter, packages need to be scored, based on the scores of the entities they consist of[2].

Entities are scored using two functions, F_{aggr} and F_{comb}. F_{aggr} is used to aggregate the scores of multiple documents that contain a specific search term and that are related to an entity. F_{comb} is then used to combine the aggregated scores of an entity for all search terms. For example, in fig. 1 assume that documents d_3, d_6 contain the term "stunning", with associated scores s_1, s_2, respectively, and document d_6 contains the term "safe", with associated score s_3. Since both these documents are related to entity Mexico City, its score wrt. keywords "stunning" and"safe" will be $F_{comb}(F_{aggr}(s_1, s_2), F_{aggr}(s_3))$.

In the following, we assume that F_{aggr} is distributive over append (i.e. $F_{aggr}(s_1, s_2, \cdots, s_n) = F_{aggr}(F_{aggr}(s_1, s_2, \cdots, s_i), F_{aggr}(s_{i+1}, \cdots, s_n))$) and subset monotonic (i.e. $\{s_1, \cdots, s_i\} \subseteq \{s'_1, \cdots, s'_j\} \implies F_{aggr}(s_1, \cdots, s_i) \leq F_{aggr}(s'_1, \cdots, s'_j)$), and that F_{comb} is monotonic (i.e. $\forall i s_i \leq s'_i \implies F_{comb}(s_1, s_2, \cdots, s_n) \leq F_{comb}(s'_1, s'_2, \cdots, s'_n)$). These properties are by no means restrictive, as they hold for most scoring functions used in practice (e.g. sum, weighted sum, max, etc.).

We note that scoring can also proceed with F_{aggr} being applied on the results of F_{comb}; this can be efficiently reduced to query processing in the framework initially described, by utilizing the TA-NRA algorithm ([10]). Using these frameworks, we are able to capture most interesting and practical semantics for scoring entities; for instance, we can require that each query keyword occurs in at least one document matching an entity, or that all such documents be considered as a single pseudo-document for the purpose of scoring the entity.

For the second task, of scoring packages based on their entities' scores, following common practice, we employ a monotonic function F_{pkg}. For instance, if for a given query the score of en-

[2]Packages are only scored based on the scores of their comprising entities, because, in the EPF setting, packages have been defined as embodying *certain* knowledge. Although our setting can be extended to "fuzzy packages" with an attached confidence value, and corresponding changes to the scoring framework, we do not further discuss such extensions in this work.

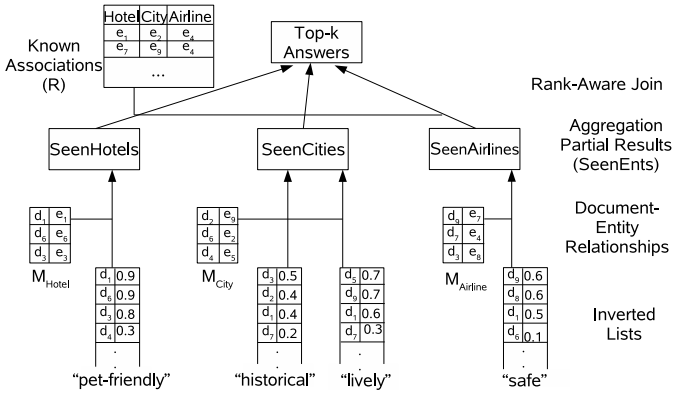


Figure 2: Execution Framework

tity Hotel Presidente is s_1, the one of Cancun is s_2, and that of Aeromexico is s_3, the score of the resulting holiday package will be $F_{pkg}(s_1, s_2, s_3)$.

2.4 A Threshold Algorithm Framework

We now present a general framework for processing entity package finder queries (fig.2).

As seen in fig. 2, processing is driven by sequential accesses on *inverted lists*, one for every keyword in the query. A list corresponding, e.g. to keyword "safe", consists of document ids for every document containing the word "safe", along with scores that denote their relevance to "safe", in descending score order. Every document id encountered is used to probe in-memory *document-entity relationship* tables (cf. $M_{Airline}$ in fig. 2), so as to retrieve the entities related to the document. The document score and related entities are used to update the current known bounds on entity scores. Such information is maintained in the Aggregation Partial Result Tables, termed $SeenEnt$s for brevity (cf. $SeenAirlines$ in fig. 2); we describe their functionality shortly.

Using the available score bounds on entities, a partial ordering on them can be determined (e.g., if entity e_1 has a score in $[1, 2]$, and the score of e_2 is in the interval $[2.3, 3.5]$, then clearly e_2 will have a higher score than e_1). Whenever a query processing algorithm decides that sufficient information wrt. entity scores is available (e.g. the top-3 Airlines, top-2 Cities and top-5 Hotels, and their scores are known), it can use this information to try and calculate the k top-scoring packages. This is done by essentially performing a multi-way join among the top entities of each type, using probes to the in-memory *known associations* view, R, to evaluate the join condition. To avoid superfluous processing effort, a query processing algorithm should use a rank-aware join operator ([11]), allowing for earlier termination, without the need to consider all top-scoring entities. Such approaches are described in sec. 3 and sec. 4.

Computing Entity Aggregate Scores: For every entity type T, we maintain a table $SeenEnt_T$ (cf. $SeenCities$ in fig. 2), which captures the current level of knowledge wrt. the most promising entities in T. Specifically, for every entity e, $SeenEnt_T$ maintains lower and upper bounds on its final score (LB and UB, respectively). These bounds are used as in the Threshold Algorithm ([10]), for pruning entities guaranteed not to be part of the final answer, and for early termination. For reasons of efficiency, $SeenEnt_T$ is indexed by entity id.

Computing Lower Bounds: In order to compute lower bound information, $SeenEnt_T$ additionally records, for every entity e, the number of times a document related to e has been encountered on every inverted list $list_i$, $NumSeen_i$, and the score that e has received up to now from all documents on $list_i$, $AgScore_i$. Put differently, $AgScore_i$ is the score that e would obtain from the keyword corresponding to $list_i$, if no other documents matching e are found on $list_i$. $AgScore_i$ is a lower bound on the final score of e wrt. the keyword corresponding to $list_i$, and the full final score of e is lower bound by $LB = F_{comb}(AgScore_1, AgScore_2, \cdots)$.

Computing Upper Bounds: In order to calculate an upper bound, UB, on the score of entity e, we utilize, for every inverted list $list_i$, the following items of information. Firstly, the score of the last document retrieved from $list_i$, $maxUnseen_i$; this is an upper bound on the score that any document, not yet retrieved from $list_i$, can have. Secondly, we require knowledge of the maximum number of documents in $list_i$ that can influence the score of e, which we term the *cardinality* of e in $list_i$, $card(e, list_i)$; we shortly discuss ways of calculating it. The maximum further score e can obtain from $list_i$ is $maxFurther_i = F_{aggr}(maxUnseen_i, \cdots, maxUnseen_i)$, where F_{aggr} is applied on $card(e, list_i)$ variables with value $maxUnseen_i$. The maximum score e can obtain from $list_i$ is $maxTotal_i = F_{aggr}($ $AgScore_1, maxFurther_1)$. Finally, an upper bound on the score of e is $UB = F_{comb}(maxTotal_1, maxTotal_2, \cdots)$.

Computing Entity Cardinalities: Let us now discuss how the cardinality of an entity e in a list $list_i$ can be calculated, by judiciously materializing information about certain entities. In a preprocessing phase, we scan all inverted lists, and calculate and store the ids and scores wrt. $list_i$ of the X entities with the highest cardinalities in $list_i$, as well as an upper bound, $maxCard_i$, on the cardinality of all other entities in $list_i$ [3]. By using this information at query time, for every entity e and list $list_i$ we have knowledge of either i) a useful upper bound on the cardinality of e in $list_i$ (namely, $maxCard_i$), or ii) the precise score of e wrt. $list_i$ (in this case, we record the score in the relevant $SeenEnt$ table, and set the relevant $NumSeen_i$ to the special value "ALL"). This enables the calculation of the maximum score, $MaxUnseen_T$, obtainable by any entity e of type T that has not yet been seen (exploiting the the monotonicity properties of F_{aggr}, F_{comb}). $MaxUnseen_T$ can be used to compute an early termination criterion. We note that this preprocessing has a low overhead[4].

2.4.1 Access Primitives

Given the structures above, we define two forms of access that a query processing algorithm can use to access data: Sequential Accesses (SA's), and Batch Accesses (BA's) (the latter correspond to standard TA Random Accesses, optimized for the entity package finder setting).

A SA on an inverted list (alg. 1) essentially retrieves a block[5] of document ids from the list (line 1), looks up entities related to these documents (line 2), and updates bounds on the scores of these entities (lines 3-8). At some point of time, an algorithm determines that a superset of all necessary entities of a type T has been identified (i.e., any entity that does not have a corresponding entry in $SeenEnt_T$ is not needed to identify the query answer). Thus, the

[3] X is a small number, so that storing these entities and their scores requires a negligible amount of storage; for instance, in our experiments we materialized 1% of all entities, resulting in an additional storage overhead of less than 0.01% the size of inverted lists.

[4] For instance, on a large, real dataset we used in our experiments (see sec. 5.2, corpus $REAL$) this preprocessing requires on average less than a second per inverted list per entity type; similar trends were observed on larger synthetic datasets.

[5] For reasons of efficiency, and following common practice, documents are retrieved in blocks, as opposed to one-at-a-time.

Algorithm 1 Sequential Access

Input: An inverted list $list_i$, corresponding to search term w, and an entity type T

1: Retrieve next block of documents and scores from $list_i$
2: Lookup entities related to these documents using M_T (In-memory join of document ids with M_T)
3: **for each** entity e found, whose related document's score is s **do**
4: **if** an entry for e exists in $SeenEnt_T$, with $NumSeen_i \neq$"ALL" **then**
5: Increment e's $NumSeen_i$
6: Update e's $AgScore_i$ with $F_{aggr}(AgScore_i, s)$, and update its LB using F_{comb}
7: **else if** e has not been pruned from $SeenEnt_T$ **then**
8: Create an entry for e in $SeenEnt_T$, with $NumSeen_i = 1, AgScore_i = F_{aggr}(s)$, and calculate its LB

Algorithm 2 Batch Access

Input: An inverted list $list_i$, corresponding to search term w, and an entity type T

1: Let $M'_T = M_T \ltimes_{EntId} SeenEnt_T$ (M'_T is not necessarily materialized)
2: Retrieve next block of documents (along with their scores) from $list_i$
3: Lookup entities related to these documents using M'_T (In-memory join of document ids with M'_T)
4: **for each** entity e found, whose related document's score is s **do**
5: **if** e's $NumSeen_i \neq$"ALL" **then**
6: Increment e's $NumSeen_i$
7: Update e's $AgScore_i$ with $F_{aggr}(AgScore_i, s)$, and update its LB using F_{comb}

only necessary actions are to discover the actual scores of the required top entities, and to prune away the remaining ones.

For this purpose, Random Accesses on the scores of all candidate entities are needed. However, if one were to perform individual Random Accesses on the score of an entity, e, one would need to i) retrieve all documents related to e, and ii) scan them to determine their score wrt. all query keywords. These operations would entail a large number of disk random seeks. For instance, in a corpus of real data we used in our experiments (sec 5.2, corpus $REAL$), on average 17-40 documents would have been read from disk per Random Access, depending on the entity's type. Hence, true Random Accesses are not a feasible option in our setting. Due to these performance considerations, we instead process Random Accesses in batches, using an access primitive we term Batch Access (BA) (shown in alg. 2).

A BA on an inverted list follows the steps we outlined above for SA's; however, it ignores documents that do not correspond to entities encountered so far. Thus, the result of a series of BA's on a list is precisely the same as that of performing random accesses on the scores of all entities wrt. the list, at a fraction of the cost.

3. THE LAYERED ALGORITHM

Given the above framework, we present the *Layered* algorithm (shown in alg. 3), a baseline approach for solving the entity package finder problem.

We define D_i as the number of top-scoring entities of a type T_i that are needed to identify the top-scoring packages. Assume that D_i are known, for all i (We will subsequently argue that this is an unrealistic assumption, and show how to alleviate it.). For instance, in fig. 2, assume it is known that, given the top D_1 Hotels, the top D_2 Cities and the top D_3 Airlines, along with their scores, it is possible to calculate the top k holiday packages. Then, the top k packages can be calculated by first identifying the top D_i entities of each type T_i, and then using these to compute the top k packages.

Algorithm 3 Layered Algorithm

Input: Query W, Number of desired answers k, Necessary number of top entities per type, $D_1, \cdots, D_n$
Output: The top k packages wrt. W

1: **for each** entity type T **do**
2: $performBA =$ **false**
3: **while** The top D entities of type T have not been identified **do**
4: **if** $performBA$ **then**
5: Perform a BA on every inverted list for type T
6: **else**
7: Perform a SA on every inverted list for type T
8: Update UB's in $SeenEnt_T$, and prune nonviable entities
9: **if** $minK_T \geq MaxUnseen_T$ **then**
10: $performBA =$ **true**
11: **loop**
12: **for each** entity type T **do**
13: Retrieve e, the next best entity e of type T
14: Retrieve from R all packages P containing e
15: **for each** package $p \in P$ **do**
16: Retrieve the scores of all entities in p.
17: **if** the score of some entity in p is not known **then**
18: Continue with the next package in P
19: Calculate $s =$ the score of p
20: **if** $s > minK$ **then**
21: Remove the top k'th package form the priority queue, and enqueue p in its place
22: **if** $minK \geq MaxUnseenPkg$ (see eqn. 1) **then**
23: **return** the current top k packages

More specifically, the Layered algorithm (a high-level outline of which is shown in alg. 3), maintains a priority queue for every entity type T_i, containing the D_i entities with the currently largest LB's. Let $minK_{T_i}$ be the smallest LB among these. Every entity of type T_i with $UB < minK_{T_i}$ can safely be pruned from $SeenEnt_{T_i}$, as it is certainly not among the top D_i entities of its type (line 8). Using this pruning criterion, Layered performs SA's on every inverted list until a superset of the top D_i entities of each type T_i have been identified (line 9); thereafter it performs BA's instead (line 10), until the top D_i entities are identified.

Subsequently, a rank-join ([11]) algorithm is used to calculate the top packages (lines 11- 23). Layered iterates over every entity type T, selecting the next best entity e of type T (line 13). It then uses e to probe the known associations table, R, and retrieve the packages containing e(line 14). For every such packages p, if the scores of all entities in p are known, Layered calculates the exact score of p (using F_{pkg}) (lines 16-19). Layered also maintains the current top k packages in a priority queue, and the score of the current top-k'th package, $minK$.

At any point of time, in order to decide whether the priority queue contains the true top k packages, and thus it is safe to terminate processing, Layered utilizes the following observation, due to the monotonicity of F_{pkg}. Let $s_i(b)$ be the score of the top b'th entity of type T_i. After having examined the top b_i entities of type T_i, the best score obtainable by a package that has not yet been encountered is

$$MaxUnseenPkg = max\{F_{pkg}(s_1(b_1), s_2(1), s_3(1), \cdots, s_n(1)), F_{pkg}(s_1(1), s_2(b_2), s_3(1), \cdots, s_n(1)), F_{pkg}(s_1(1), \cdots, s_{n-1}(1), s_n(b_n))\} \quad (1)$$

Layered terminates when $minK \geq MaxUnseenPkg$ (line 22).

3.1 Execution example

To better illustrate the workings of Layered, we present the following execution example. We will utilize the sample corpus shown in table 3.1, and the following scoring functions: $F_{aggr}(x_1, \cdots, x_j) =$

(a) Inverted List for keyword w_1

DocId	Score
d_5	1.0
d_3	0.8
d_6	0.5
d_8	0.2
d_7	0.2

(b) Inverted List for keyword w_2

DocId	Score
d_1	1.0
d_7	0.9
d_4	0.6
d_2	0.2
d_9	0.1

(c) Inverted List for keyword w_3

DocId	Score
d_{13}	0.7
d_{18}	0.5
d_{15}	0.2
d_{16}	0.1
d_{10}	0.1

(d) Doc - Entity relationships, M_{T_1}

DocId	EntId	DocId (cont'd)	EntId
d_1	b	d_6	e
d_2	b	d_7	a
d_3	a	d_7	c
d_4	a	d_8	b
d_5	a	d_8	c
d_5	b	d_9	c
d_6	c	d_9	e

(e) Doc - Entity relationships, M_{T_2}

DocId	EntId
d_{10}	α
d_{10}	γ
d_{13}	α
d_{15}	γ
d_{16}	α
d_{18}	β

(f) Known associations, R

Entity type T_1	Entity type T_2
b	α
a	γ
b	β
e	β

(g) Layered execution example, $SeenEnt_{T_1}$

Step	EntId	NumSeen w_1	AgScore w_1	NumSeen w_2	AgScore w_2	LB	UB
1	a	1	1	0	0	0	3
	b	1	1	1	1	1	3
2	a	2	1.8	1	0.9	0.9	2.6
	b	1	1	1	1	1	2.6
	c	0	0	1	0.9	0	2.4
3	a	2	1.8	2	1.5	1.5	2.1
	b	1	1	1	1	1	2
	c	1	0.5	1	0.9	0.5	1.5
	e	1	0.5	0	0	0	1.5
4	a	2	1.8	2	1.5	1.5	1.7
	b	2	1.2	2	1.2	1.2	1.4
	c	2	0.7	1	0.9	0.7	0.9
	e	1	0.5	0	0	0	0.6

Table 1: Sample corpus and Layered execution example

$\sum x_l$, $F_{comb}(x_1, \cdots, x_j) = min(x_l)$, $F_{pkg}(x_1, \cdots, x_n) = \sum x_l$ if $min(x_l) > 0$, and 0 otherwise. For the sake of clarity, we assume that, during preprocessing, no entity scores were materialized, and the bound computed on the maximum cardinality of all entities wrt. any list is 3 (i.e., in this corpus, the score of every entity can be influenced by at most 3 documents). Moreover, we assume that SA's access inverted lists one document id at a time, even though in practice, for performance considerations, document id's are retrieved in larger batches.

Assume we want to identify the top 1 package wrt. query $(\{w_1, w_2\}, \{w_3\})$; i.e., the top package where the first entity (of type T_1) is most relevant to keywords w_1 and w_2, and the second entity (of type T_2) is most relevant to w_3. Moreover, assume Layered has somehow determined that $D_1 = 3$ and $D_2 = 2$ top entities of each type, T_1 and T_2, are needed in order to determine the top 1 package.

Layered will first identify the top $D_1 = 3$ entities of type T_1 (lines 3-10). A SA is performed on both inverted lists of w_1 and w_2 (line 7). This results in documents d_5 and d_1 being retrieved. By probing M_{T_1}, Layered discovers that they are related to entities a, b and b, respectively. Entries for a and b are created in $SeenEnt_{T_1}$; information regarding how many times they have been encountered on every list, and their aggregate score wrt. every list, is recorded. a initially has lower bound $LB = F_{comb}(1, 0) = 0$, since, at this point, it is possible that its score wrt. w_2 will be 0 (recall $F_{comb} = min$). Upper bounds for all entries in $SeenEnt_{T_1}$ are then calculated, (line 8) using their current $AgScore$'s, and every $maxUnseen_i$ (which is currently 1, for both lists). The state of $SeenEnt_{T_1}$ after these operations is shown in table 0(g), step 1. This procedure is repeated another three times (tab. 0(g), steps 2-4). Moreover, after step 4, the best score that entity e can possibly obtain ($UB = 0.6$) is less than the worst possible score that any of the top-3 entities a, b or c can obtain; as only 3 top entities need to be calculated, e is pruned (line 8). Independently, since no further entity yet unseen can become part of the top-3 (line 9), Layered proceeds by performing BA's on all lists, until the final scores of a, b and c (1.5, 1.2 and 0.9, respectively) are discovered. Similarly, Layered identifies that the top $D_2 = 2$ entities of type T_2 are α and β, with scores 0.9 and 0.5, respectively.

Layered then proceeds by joining the identified top entities of types T_1 and T_2, using known associations table R, in a rank-aware fashion (lines 11-23). Using R, after processing entities a, b and α, Layered determines that b, α is the currently top package, with score 2.1. Moreover, the next best entities of each type have scores 0.9 and 0.5, respectively. Thus, assuming they were associated with the top entities of the other type, the resulting packages would have scores $F_{pkg}(0.9, 0, 9) = 1.8$ and $F_{pkg}(1.5, 0, 5) = 2$, respectively. Since none of these is better than the score of the package b, α, Layered concludes that it is the top 1 package, and terminates processing (line 22).

3.2 Estimating D_i

The Layered algorithm crucially depends on knowing the precise value of all D_i (the number of entities of every type T_i that are needed, in order to identify the top k packages). Clearly, this is an unrealistic assumption, as in general one would need to execute the entity package finder query in its entirety in order to determine these values. Thus, a realization of the Layered algorithm would call for an *estimation* D'_i of D_i. Let us now examine the implications on query processing, by showing how the D'_i estimation can be performed using state-of-the art techniques.

This estimation problem is strongly related to the Depth Estimation problem in the context of a rank-join algorithm ([16]). Indeed, [16] proposes techniques that could be readily adapted for estimating D'_i. However, this adaptation necessitates a strong assumption; namely that sufficient statistics are known about the distribution of the scores of packages wrt. any given query. The statistics should be detailed enough to provide the following information, as accurately as possible:

Given a tuple of entity scores $(s_1, s_2, \cdots, s_n)$,

- How many valid packages $(e_1, e_2, \cdots, e_n)$ exist, such that the score of each e_i is precisely s_i? and
- For every i, what is the largest number s'_i such that $s'_i < s_i$, and there exists at least one valid package with score $(s_1, \cdots, s_{i-1}, s'_i, s_{i+1}, \cdots, s_n)$?

In order to provide such statistics, one has essentially two options. Firstly, one can materialize in a preprocessing phase, for every entity of every type, the precise distribution of the scores of its related documents wrt. every inverted list. These distributions can then be used, in conjunction with the known associations table, to estimate the score distribution of packages at runtime. However, materializing all this information has an enormous space overhead,

on the order of the corpus size itself[6].

The second option, is to estimate the score distribution of packages by sampling (assuming the API available on R enables such an operation, e.g. in the case of dynamic known associations). Specifically, at runtime, one can select some packages at random, and calculate exactly the scores of all their entities, wrt. the given query. Unfortunately, these calculations have a very high runtime overhead, as they require *Random Accesses* on the scores of a number of entities, which are unfeasibly expensive operations in our setting[7]). Moreover, in order to ensure that the sampling produces a reasonably small estimation error, a sizeable fraction of entities' scores would need to be calculated. Hence, the runtime overhead of this option is clearly too high to be used in practice.

Summarily, using state-of-the art techniques for estimating D'_i in the context of the Layered algorithm, will either have an excessive storage and/or runtime overhead, or will entail sizeable and unpredictable estimation errors. These errors will largely affect the performance of the Layered algorithm. If, for some i, $D'_i > D_i$, Layered will be forced to perform more processing than necessary. Even worse, if for some i, $D'_i < D_i$, Layered will not be able to identify the top k packages; thus it will need to perform some additional processing (including re-doing some processing previously performed), to first identify a number $D''_i > D'_i$ of top entities of type T_i, and then resume the rank-join process.

It should thus be clear that an efficient realization of Layered is not possible, due to the significant overheads imposed by the estimation procedure. To address this issue, we subsequently introduce Interleaved, an efficient algorithm for the entity package finder problem, that *does not rely* on knowledge or estimations of D_i. Despite not using such estimates, we show that Interleaved is as efficient as a hypothetical, idealized instantiation of the Layered algorithm, that would have freely available accurate knowledge of all D_i.

4. THE INTERLEAVED ALGORITHM

The essense of the Interleaved algorithm is to incrementally maintain score information on the package level. To do so, while scanning inverted lists, Interleaved keeps updating the score bounds of both entities, as well as packages (by interleving the operations of rank-aware aggregation and rank-join). Using this knowledge, it is able to prune non-promising entities and packages, and thus to terminate early. The high efficiency of Interleaved originates principally from two novel features: the *interleaved pruning* of entities, and the *tight bounding* of package scores. Both these features are designed to exploit the distribution of fixed associations, and are subsequently explained in detail. Through careful engineering, this added functionality is achieved with only moderate extra bookkeeping effort.

Algorithm 4 Interleaved Algorithm

Input: Query W, Number of desired answers k
Output: The top k packages wrt. W
1: **while** the top k packages have not been identified **do**
2: $performBA =$ **false**
3: **for each** entity type T **do**
4: **if** $performBA$ **then**
5: Perform a BA on every inverted list for type T
6: **else**
7: Perform a (modified) SA on every list for type T
8: Update UB's in $SeenEnt_T$
9: **for each** entity type T **do**
10: Update UB's in $Cand$, using information from $SeenEnt_T$
11: Prune packages with $UB < minK$, and decrement the $refCount$s of all their entities
12: Update LB's in $Cand$ in packages with entities whose LB's have been updated
13: Prune entities with $refCount = 0$
14: **if** $minK \geq MaxUnseenPkg$ (see eqn. 2) **then**
15: $performBA =$ **true**

Interleaved maintains a few additional data structures, besides those detailed in section 2.4. Firstly, a *Candidate Package* table, $Cand$, which contains all promising packages (i.e. those that might rank among the final top-k). Entries in $Cand$ are of the form $(e_1, e_2, \cdots, e_n, UB, LB)$, where e_i denote pointers to the entry of entity e_i in the respective $SeenEnt_{T_i}$ table; UB and LB are bounds (upper and lower, respectively) on the final score of the package. Interleaved also maintains a priority queue containing the k most promising packages (based on their LB's), as well as $minK$, the LB of the k'th top package. Finally, entries in $SeenEnt$ tables are augmented with a *reference count* field, $refCount$ (this field is used for interleaved pruning, and is described shortly).

Let us now examine the operation of Interleaved (pictured in alg. 4) in detail. Until the top-k packages have been identified, Interleaved does the following (line 1). First, it performs some SA's on every inverted list, for every entity type (line 7). The algorithm for SA's it uses is slightly modified from the alg. 1. Specifically, whenever an entity e is first encountered (line 7 in alg. 1), entries in $Cand$ are created for all the packages e participates in. This is done by probing the known associations table, R, for packages containing e, and creating $Cand$ entries for every such package. If needed, placeholder entries are created in $SeenEnt$ tables for other entities in these packages; subsequently, the $refCount$s of all entities in these packages are incremented.

Having performed these modified SA's, Interleaved updates upper bound information in $SeenEnt$ tables (line 8). Using this information, it then updates upper bounds in the Candidate Package table (line 10). Interleaved prunes packages that are certain to not rank among the final top-k, based on their score upper bound, and decrements their entities' $refCount$s (line 11). Interleaved also updates lower bound information in $Cand$ (line 12); for the sake of efficiency, it only updates packages containing entities with recently updated LB's (this is recorded in a separate flag).

In line 13, the feature of interleaved pruning manifests itself. Entities that do not participate in any promising package are pruned. In this manner, a large number of entities can be pruned, independently of how high their score is, just by exploiting the distribution of known associations. This leads to a significant decrease in processing effort, as there are fewer entities to keep track of, and to earlier termination. Moreover, by keeping track of the number

[6]In a large-scale, real dataset utilized in our experiments (see sec. 5.2, corpus $REAL$), storing such information incurs an additional average space overhead of 15% per entity type. The corpus contains 9 entity types, so these statistics require more storage than the corpus itself. A similar space blowup was observed in the synthetic corpora we utilized in our experiments.

[7]With respect to the cost of Random Accesses, note that, in order to calculate the score of a single entity, one would need to retrieve all documents in the corpus that are related to the entity, and scan them to determine their score wrt. all query keywords. Thus, this calculation entails a number of disk *random seeks* equal to the cardinality of the entity (i.e. the number of documents related to it); in a corpus of real data we used in our experiments (sec 5.2, corpus $REAL$), this was on average 17-40 documents per entity (depending on the type of the entity). Note also that such Random Accesses cannot be performed in batches - i.e. using Batch Accesses- as this would result in redundantly executing a large part of the query itself in a preprocessing phase, solely for the purpose of obtaining estimates.

of packages an entity participates in (through its $refCount$), interleaved pruning is very efficient, and imposes but a minimal storage overhead.

Finally, in line 14, Interleaved decides whether a superset of the final top k answers has been identified, and SA's can therefore be replaced by BA's. Another significant feature of Interleaved is an improved condition for making this decision. Adapting standard rank-join techniques ([11]), as in the case of Layered, results in an overly conservative bound on the maximum score of a package not yet encountered. Specifically, let $s_i(b_i)$ be the best score obtainable by an entity of type T_i that has not yet been encountered, and let $s_i(1)$ be the overall best score obtainable by an entity of type T_i. Then, the bound on the best score obtainable by a package that has not yet been encountered is given by eqn. 1. This bound corresponds to assuming that a package may exist, containing the top entity of each type but one, and an entity yet unseen on that type. Interleaved, on the other hand, by maintaining bounds on the scores of all packages that contain at least one seen entity, has access to more information; namely that any package not in $Cand$ can only consist of entities not yet encountered. Hence, Interleaved uses the following upper bound on the score of a package not in $Cand$ in line 14:

$$MaxUnseenPkg = F_{pkg}(MaxUnseen_{T_1}, \cdots, MaxUnseen_{T_n}) \quad (2)$$

This bound, which exploits known associations, is in fact a *tight bounding* scheme on package scores (in the spirit of [15]); in a sense, this is the tightest possible bound that can be provided. More formally, there always exists a set of documents that could appear later on some inverted lists, and a set of their relationships to entities in the corpus, such that the most promising package not currently in $Cand$ obtains a final score equal to $MaxUnseenPkg$. This is in contrast to the bound employed by Layered and by rank-join algorithms, which can be significantly looser. The tight bounding scheme significantly boosts the pruning power of Interleaved.

To summarize, in Interleaved we have effectuated an early pruning and termination strategy, that is entirely independent of any estimates, and has a very moderate bookkeeping overhead. Entities are now pruned based on whether they might be part of some top package, rather than their current score alone, by dint of an *interleaved pruning* policy that exploits the distribution of packages. Moreover, in contrast with standard rank-join algorithms, package scores are *tightly bounded*, based on the best score they can actually achieve. These features offer the Interleaved algorithm additional opportunities for improved performance, compared to Layered, without relying on potentially erroneous and expensive estimations.

4.1 Execution example

To illustrate the workings of Interleaved, we show how it processes the scenario previously discussed in sec. 3.1; we refer to table 3.1 for the corpus used. In table 4.1 we provide a trace of the contents of the $Cand$ table at every execution step (i.e., at every iteration of line 1). Moreover, as the functionality of $SeenEnt$ tables has been demonstrated in sec 3.1, we only provide a summary thereof, containing LB's and UB's for every entry. In step 1, a SA is performed on every inverted list (line 7), and $SeenEnt$ tables are populated. Moreover, entries concerning all packages that contain an entity in some $SeenEnt$ table are inserted in $Cand$. Finally, empty entries are inserted in $SeenEnt$ tables, concerning entities that are mentioned in some package in $Cand$, but do not have a $SeenEnt$ entry (e.g. β, γ). This procedure is repeated in step 2.

	$SeenEnt_{T_1}$			$SeenEnt_{T_2}$			$Cand$			
Step	EntId	LB	UB	EntId	LB	UB	$EntId_1$	$EntId_2$	LB	UB
1	a	0	3	α	0.7	2.1	b	α	1.7	5.1
	b	1	3	β	0	2.1	b	β	0	5.1
				γ	0	2.1	a	γ	0	5.1
2	a	0.9	2.6	α	0.7	1.7	b	α	1.7	4.3
	b	1	2.6	β	0.5	1.5	b	β	1.5	4.1
	c	-	-	γ	0	1.5	a	γ	0	4.1
	e	0	2.4				e	β	0	3.9
3	a	1.5	2.1	α	0.7	1.3	b	α	1.7	3.3
	b	1	2	β	0.5	0.9	b	β	1.5	2.9
	e	0	1.5	γ	0.2	0.6	a	γ	1.7	2.7
							e	β	0	2.4
4	a	1.5	1.7	α	0.8	0.9	b	α	2	2.3
	b	1.2	1.4	β	0.5	0.7	b	β	1.7	2.1
	e	0	0.6	γ	0.2	0.4	a	γ	1.7	2.1
							e	β	0	1.5

Table 2: Interleaved Execution example

Note that entity c was pruned (line 13), as it does not participate in any package. After 2 more steps, in step 4, package e, β is ascertained to not be the top package, and it is pruned. This results in the pruning of entity e as well. In the final step (not shown), package scores are fully disambiguated, and package b, α is determined to be the top package with score 2.1.

4.2 Extensions

Informed Access Scheduling: The Interleaved algorithm described above is very efficient, without requiring any estimations. However, it is still not optimal. Consider a problem instance where, in order to identify the top 5 packages, 500 SA's are required on one inverted list, and only 10 on all the others. Since Interleaved performs SA's in a round robin fashion, 500 SA's will be performed on all lists before the algorithm terminates. It could thus help performance if accesses were scheduled taking properties of the underlying data distribution into account, so that fewer accesses are performed till termination.

Techniques for scheduling accesses in an informed manner have been proposed in [7], in the context of standard top-k query processing. It is only natural to investigate the extent to which such methods may be beneficial to our setting as well. Utilizing such techniques, one could try to effectively allocate SA's to inverted lists, so as to reduce the total number of required accesses. Using statistics on i) the score distribution of every inverted list and ii) the distribution of documents on every inverted list matching every entity, one may try to adapt the KBA framework from [7] to the entity package finder setting. Periodically, the adapted KBA framework can be used to estimate the "Benefit" that some number of SA's on a given list will have on processing, and select the most "beneficial" allocation of SA's to inverted lists ("Benefit" here is a quantity highly correlated with the remaining processing effort, i.e. highly "beneficial" SA's lead to shorter expected processing times).

Optimizing the schedule of SA's requires posing a number of queries over the materialized statistics. Due to our setting involving aggregation, compared to the setting in [7], informed scheduling requires that orders of magnitude more such queries be posed. Moreover, it requires that additional independence assumptions be made, leading to lower accuracy; hence it is expected to be less useful in reducing processing overhead. These expectations are consistent with our empirical observations, on our adaptation of KBA. On average, more than half of query time was spent on informed schedul-

ing, as opposed to actual query processing; moreover, the amount of query processing effort was not significantly reduced. For these two reasons, we do not further pursue an informed scheduling approach in this work. Further details on our adaptation of KBA can be found in [3].

Further extensions: More generally, we note that Interleaved is amenable to further extensions of the entity package finder problem, such as taking into account individual user preferences; we discuss such extensions in [3].

5. EVALUATION

5.1 Analysis

As introduced, the Interleaved algorithm has two potential sources of performance benefits: *interleaved pruning* of entities and *tight bounding* of package scores. In this section, we propose two models that capture the intuition behind these sources, and help in comparing Interleaved with Layered. To effectuate this comparison, we use a hypothetical, idealized instantiation of the Layered algorithm, by providing it with an oracle for the exact values of all D_i for the query being processed. To highlight this fact, we refer to this oracle instantiation of Layered as Layered^{D_i}.

5.1.1 Benefits of interleaved pruning

To capture the benefits of interleaved pruning, we introduce the notion of an *entity-package ranking coefficient* of a dataset. Most top-k query processing approaches (recast in entity package finder terms) assume that top packages tend to be composed of top entities. The entity-package ranking coefficient embodies, in a sense, the quantitative relationship between these; i.e. how many top entities are needed for every top package. As we will shortly see, this relationship strongly affects the effectiveness of interleaved pruning.

Whereas top packages tend to be composed of top entities, the converse is clearly not true; an entity e with a mediocre score can be part of a top package, and an entity e' of the same type, with a higher score than e, might not be part of any. Let E_i denote the set of all entities of the latter kind wrt. a query (i.e. top entities of type T_i, which nevertheless do not participate in any top package). Layered^{D_i} expends unnecessary effort to fully process entities in E_i , by computing the top D_i entities of each type *independently* of the resulting packages. Interleaved pruning, however, provides Interleaved the ability to avoid this overhead. We therefore expect that, the more entities in all E_i that exist wrt. a query, the greater the performance improvements due to interleaved pruning. Observe that the number of such entities is strongly correlated with the total number of top entities, D_i, that need to be examined, as only a fraction of all entities examined will participate in some top package. Finally, note that all D_i are positively correlated with the number of top packages required, since the more top packages that need to be identified, the more entities will have to be examined. Thus, we can quantify the effects of interleaved pruning by examining the expected number of entities that need to be examined per answer package identified, i.e. the expected value of $\frac{D_i}{k}$.

We quantify this value as follows: For a given corpus, and for every entity package finder query W, let k be the number of top packages requested by W. As per definition, the rank of the lowest scoring entity of type T_i that needs to be computed, in order to identify the top k packages, is D_i. We define the entity-package ranking coefficient for this case, $c_i(T_i, W, k)$, as $c_i(T_i, W, k) = \frac{D_i}{k}$. Essentially, c_i provides a measure of the number of entities of type T_i that need to be calculated, in order to identify an additional top package; in other words, a measure of the necessary amount of processing per result package. In general c_i will depend on many factors, such as the corpus, the query being processed, and the number of top packages requested.

Let $P(W, k)$ be the probability that query W is issued, requesting a number k of top packages (this probability may be derived from a known querylog, or via a uniformity assumption). We define the *entity-package ranking coefficient* of our entire dataset C, as a random variable, distributed according to the following distribution:

$$Prob(C = x) = \frac{1}{n} \sum_{\forall T_i, W, k} P(W, k) \cdot [|c_i(T_i, W, k) = x|]$$

where $[|A|] = 1$ if A is true, and 0 otherwise

Given the discussion presented above wrt. interleaved pruning, we expect that in every dataset, the relative performance of Interleaved versus Layered^{D_i} to be positively correlated with the expected value of C. That is, for larger expected values of C, we expect the performance gains of Interleaved versus the idealized Layered^{D_i} to increase, due to the effects of interleaved pruning. We validate this trend in sec. 5.2, using *micro-experiments* which measure local expected values of C in a dataset.

5.1.2 Benefits of tight bounding

We subsequently introduce the notion of *entity association probability*, that illuminates the performance benefits due to tight bounding. Entity association probability embodies a notion of known association "density", i.e. what fraction of possible associations in fact hold. We shortly discuss how this measure captures the performance benefits of tight bounding.

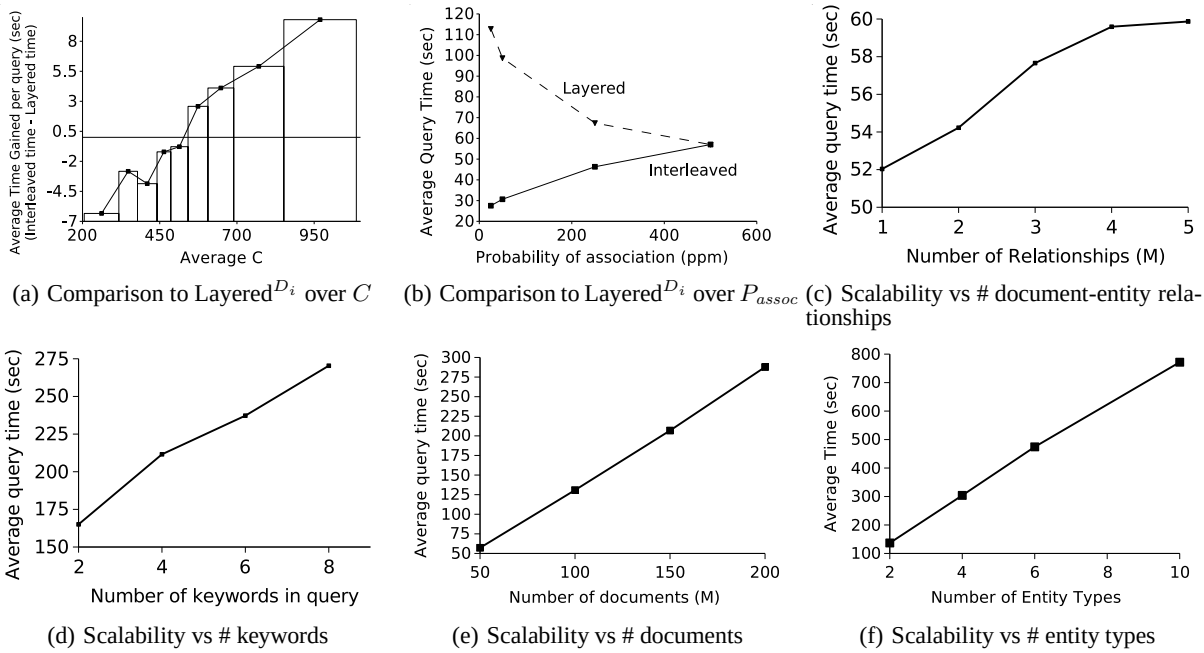
Recall that Interleaved utilizes known associations to provide tight bounds on the maximum score a package not yet encountered can have, $MaxUnseenPkg$. Tight bounding results from ensuring that the best package not yet encountered only consists of entities not yet encountered. Thus its score will be an aggregate of the scores, $MaxUnseen_{T_i}$, of these entities (eqn. 2), as opposed to an aggregate of the scores of the *best entities of each type* and one of $MaxUnseen_{T_i}$ (eqn. 1). The tightness of these bounds, and hence the effectiveness of the tight bounding scheme, depend on the probability, P_{assoc}, that n arbitrary entities of different types are associated (i.e. form a valid package). As this likelihood increases, the value of information obtainable by known associations decreases, and tight bounding approaches the bounding scheme used by Layered^{D_i}. To illustrate this point, observe that knowing which packages are valid is not very useful for bounding if a large fraction of possible packages are valid. Note that, by definition, $P_{assoc} = \{\#\text{ of known associations}\}/\Pi_i\{\#\text{ of entities of type } T_i\}$

On the other hand, Layered^{D_i} bounds $MaxUnseenPkg$ in a conservative manner, as it cannot exploit known associations. Moreover, the number of top entities it calculates per type, D_i, is not dependent on the resulting packages. Thus, by increasing P_{assoc}, the only component affected in Layered^{D_i} is the rank-aware join operation. The latter is expected to terminate earlier for larger P_{assoc}, as P_{assoc} corresponds to the join selectivity.

Thus, we expect the relative performance of Interleaved versus Layered^{D_i} to be negatively correlated with P_{assoc}; i.e. for smaller values of P_{assoc}, we expect the performance gains of Interleaved versus the idealized Layered^{D_i} to increase, in part due to tight bounding. We validate this trend in sec. 5.2.

5.2 Experimental Evaluation

The Interleaved algorithm processes entity package finder queries without relying on knowledge or estimates of D_i. Layered requires



(a) Comparison to Layered^{D_i} over C (b) Comparison to Layered^{D_i} over P_{assoc} (c) Scalability vs # document-entity relationships

(d) Scalability vs # keywords (e) Scalability vs # documents (f) Scalability vs # entity types

Figure 3: Experimental Evaluation

these values, and their estimation, as shown in sec. 3.2, incurs an unreasonably high runtime and/or storage overhead. To effectuate a comparison between the two algorithms, and demonstrate the performance benefits of our proposed techniques, we compared Interleaved with Layered^{D_i}, an idealized instantiation of Layered (i.e. Layered with an oracle for the exact values of all D_i). In order to implement the latter, the exact values of D_i for every query were precalculated in a brute force manner.

Our experiments, on large-scale synthetic datasets show that Interleaved is about as efficient, and in practical cases even more efficient, than this idealized instantiation of Layered, thus manifesting the effectiveness of our techniques. Moreover, they validate the trends predicted in sec. 5.1. Further experiments demonstrate the efficiency and scalability of our approach, on both real and synthetic datasets.

In order to stress our algorithm on large-scale datasets, we generated large-scale synthetic corpora, denoted $SYNTH$. Unless otherwise noted, all $SYNTH$ corpora contained 50M documents, two types of entities, 10K entities of each type and 5M document-entity relationships per entity type. Every keyword appeared in a varying fraction of all documents, ranging in 0.5-0.05. The queryload used, unless otherwise noted, consisted of 50 queries with 1-3 keywords per entity type, requesting the top 1-10 answers. For generating these corpora, all data distributions (document scores, document-entity relationships, entity associations, query sizes, etc.) were uniform. Our experiments used static entity associations, however we note that using dynamic associations, as described in sec.2.1 does not affect our results; this is due to entity associations being accessed via the same API, regardless of whether they are static or dynamic.

To demonstrate the applicability and efficiency of our approach, we also experimented on a large corpus of real data, $REAL$. We utilized data from BlogScope ([6]), an analysis and visualization tool for the blogosphere, currently monitoring over 28M blogs and over 400M blog posts. We used all indexed posts made in the 10-day period between June 11th and 21st (excluding spam, and posts in languages other than English), resulting in over 3.7M documents. From these documents, we extracted 600K Named Entities of 9 different types, using an Information Extraction tool developed in-house at the University of Toronto. In this way we extracted over 4.1M document-entity relationships. Entity associations were determined to hold between pairs of entities with statistically significant co-occurences in these blog posts, for a total of 110K associations. Whereas $REAL$ represents only a 10 day sample of discussions in the blogosphere, it serves to demonstrate that our approach can be efficiently applied on real-world, large-scale corpora.

We implemented both Layered^{D_i} and Interleaved in Java 1.6. As scoring functions (sec. 2.3) we used $F_{agg} = \sum$, $F_{comb} = min$, $F_{pkg} = \sum$; this choice of scoring functions favors packages with entities that are, on average, most relevant to all query keywords. We note that other choices of scoring functions, embodying different semantics, are possible (subject to the loose constraints detailed in sec. 2.3); experiments with different scoring functions yielded similar performance trends. Our implementation maintained document-entity relationships and known associations in main memory; as discussed in sec 2.1 this is a reasonable assumption, even for very large corpora. All our experiments were executed on a machine with an Intel Core2 Duo CPU, operating at 2.93GHz, and 4GB of memory; our experiments utilized only one CPU core. In all experiments we report query running time, measured from the moment a query is issued until answers are reported to the user. We do not take into account the small runtime overhead of system initialization, as it occurs only once, regardless of the number of queries processed.

For validation purposes, we also compared our algorithms with an approach that used only RDBMS technology. Specifically, we stored the $SYNTH$ corpus as indexed tables in a relational database (MySQL 5.5), and wrote our entity-package finder queries in SQL. We expect this approach to be highly inefficient, as it needs to calculate precise scores of every entity and package, followed by a selection of the top-k packages. Indeed, when executing our test queries, this approach had average query execution time up to an

order of magnitude larger than our proposed Interleaved algorithm, depending on query parameters. For this reason, we do not further consider such RDBMS-based approaches, but focus instead on approaches with early termination and pruning properties (Interleaved and Layered^{D_i}).

5.2.1 Comparison with Layered^{D_i}

We first present an evaluation of the relative performance of Interleaved and an instantiation of Layered utilizing an oracle to obtain precise D_i values, termed Layered^{D_i}. We stress that this algorithm (Layered^{D_i}) is provided as a point of comparison, and is not practically realizable (in practice, obtaining precise values of D_i is not possible, and estimating them incurs unreasonable storage and/or runtime overheads, see sec.3.2). In our first comparative experiment, we varied the number of known associations in the $SYNTH$ corpus, from 2.5K to 500K, corresponding to P_{assoc} of $25 \cdot 10^{-6}$ to $500 \cdot 10^{-6}$, and executed a queryload of 50 queries, each with 1-5 keywords per entity type, using both Interleaved, and Layered^{D_i} (Recall that P_{assoc} denotes the probability that n arbitrary entities of different types are associated. Due to its semantics, we expect it to have very low values in practice. To provide a sense of perspective with respect to the values of P_{assoc} tested, we note that observed values for P_{assoc} in $REAL$, the real-world corpus we used, ranged from $4.2 \cdot 10^{-6}$ to $21 \cdot 10^{-6}$. Thus, this experiment stresses Interleaved well beyond the operational parameters we typically expect to encounter.). In fig. 3(b) we show average query time for both Layered^{D_i} and Interleaved. As one can see, Interleaved outperforms Layered^{D_i} for all practical values of P_{assoc}, with performance gains of up to 76%. Moreover, one can observe the trends predicted in sec. 5.1.2, namely that the performance of Interleaved increases, and that of Layered^{D_i} decreases, for lower values of P_{assoc}, due to the effects of tight bounding. Finally, we observe a roughly equal performance (Interleaved being 0.18% slower than Layered^{D_i}), for $P_{assoc} = 500 \cdot 10^{-6}$. Even though this is an unreasonably high value (in view of the observed $P_{assoc} \in [4.2 \cdot 10^{-6}, 21 \cdot 10^{-6}]$ in $REAL$, as discussed above), we subsequently focus on it, to evaluate the performance benefits of interleaved pruning.

In our second comparative experiment, we evaluate the performance benefits of interleaved pruning. We expect Interleaved to relatively outperform Layered^{D_i} due to interleaved pruning, in cases with higher entity-package ranking coefficient, C, i.e. when top packages also require entities that are not among the top (see sec. 5.1.1). To validate this expectation, we utilized the $SYNTH$ corpus, and executed a queryload of 500 queries, each with 1-5 keywords per entity type, using both Interleaved, and Layered^{D_i}. As previously noted, the parameter P_{assoc} was chosen to ensure a roughly equal performance, on average, of Interleaved and Layered^{D_i}. Each query execution is a *micro-experiment*, where local expected values of C in the dataset can be measured (as $avg_i\left(\frac{D_i}{k}\right)$, where averages are computed over all entity types T_i, for the given query). We measured the performance benefits of interleaved pruning as Time Gained, which we define as {execution time using Interleaved} − {execution time using Layered^{D_i}}. We grouped our observations using an equi-depth histogram on the local measured values of C, and report average Time Gained per query in fig. 3(a). We observe the trend predicted in sec. 5.1.1, thus validating our previous analysis. As a note, recall that we set P_{assoc} to a "break-even" point between Interleaved and Layered^{D_i}; had we set it to a lower value, such as observed in real data, fig. 3(a) would be completely favorable towards Interleaved.

5.2.2 Scalability

Having shown significant performance benefits of our proposed algorithm, Interleaved, over the oracle baseline Layered^{D_i}, we next evaluate its scalability and efficiency, using large-scale synthetic corpora. Note that these are significantly larger, wrt. all operating parameters, than corpora one would expect in practice; as we subsequently demonstrate, using a real dataset, performance in practice is orders of magnitude better (sub-second average query time - cf. sec. 5.2.3).

Number of document-entity relationships: The number of document - entity relationships in a corpus is an important factor affecting query processing performance, as it influences the early pruning/termination capabilities of Interleaved, wrt. document score aggregation. To test the scalability of our approach, in this experiment we varied the number of document-entity relationships in the $SYNTH$ corpus, from 1M to 5M per entity type, and executed our typical query workload using Interleaved. We show average query execution time in fig. 3(c), and observe that Interleaved gracefully scales to a large number of document-entity relationships.

Number of keywords in query: In this experiment we used the $SYNTH$ corpus, and executed four query workloads of 50 queries each, using Interleaved. We varied the number of keywords in each query from 1 to 4 keywords per entity type (i.e. between 2 to 8 keywords per query). Average query execution time, shown in fig. 3(d), demonstrates that Interleaved scales gracefully with respect to the number of keywords in a query. We note that, typical user keyword queries involve a small number of keywords, a trend that we expect carries across to entity-package finder queries; in practice, we expect a typical workload to involve fewer keywords per query than in this experiment.

Number of documents: The number of documents contained in a corpus naturally affects performance, but is less crucial to performance than other parameters. Observe that, all other things being equal, scaling the number of documents in a corpus will increase I/O overhead, and the number of probes to document-entity relationship tables, but will not significantly affect other query processing components (e.g. rank-aware aggregation or join). We verify this trend by varying the number of documents in the $SYNTH$ corpus, from 50M to 200M per entity type, and executing our typical query workload using Interleaved. Fig. 3(e) shows average query processing time, demonstrating a graceful, near-linear scaleup trend with respect to the number of documents in the corpus. This trend validates our expectations that Interleaved can efficiently scale up to very large document collections.

Number of entity types: In this experiment we varied the number of entity types in the $SYNTH$ corpus, from 2 to 10, and executed our typical query workload using Interleaved. We observed a near-linear scaleup in average query execution time (shown in fig. 3(f)), demonstrating the scalability of our approach wrt. the number of entity types involved in a query. Note that, intuitively, an actual user query is expected to involve only a small number of types, and certainly fewer than 10; as in previous experiments, we chose to stress our algorithm with operating parameters significantly larger than in practice, to observe its trends wrt. scalability.

5.2.3 Experiments with real data

To demonstrate the applicability of our techniques on real data, we also utilize the $REAL$ corpus described above. For every different kind of pairwise entity associations (e.g. Person A is associated with Company C, e.g.2 Band B is associated with Person D), we executed a query workload containing 200 queries, each with 1-5 keywords per entity type, using both Layered^{D_i} and Interleaved. Query keywords were randomly chosen from a list of adjectives most commonly used in English.

The relative performance of Layered^{D_i} and Interleaved on real data validates our expectations from synthetic data (fig. 3(b)). Specifically, given that the $REAL$ corpus exhibits values of P_{assoc} significantly lower than those shown in fig. 3(b), we expect Interleaved to outperform Layered^{D_i} by a large margin. Indeed, when executed on the $REAL$ corpus, Interleaved was more than one order of magnitude faster than Layered^{D_i}. Moreover, Interleaved processed each query in under 1.5 sec; average execution time ranged from 0.25 sec to 0.5 sec per query, depending on the scenario of pairwise associations being tested. Overall, average query processing time using Interleaved was under 0.34 sec. We observe that our proposed algorithm is able to efficiently answer entity package finder queries on large, real-world corpora, validating our observations obtained from experimentation on synthetic corpora.

6. RELATED WORK

The entity package finder problem, presented in this work, belongs to the general area of top-k query processing. However, standard top-k techniques (e.g. [10]) do not apply, due to the document score aggregation that needs to take place. [9] and [5] proposed algorithms for calculating top-k over aggregation; however, these do not consider joins, and hence cannot be used for solving the entity package finder problem. Moreover, the techniques presented therein cannot be efficiently adapted to our problem, as the resulting algorithm would rely on estimations with very high runtime and/or storage overhead. Such estimation problems, albeit in much simpler settings, are discussed in [16]. The techniques proposed in this work cannot be efficiently applied in the entity package finder setting; the reason is that document score aggregation introduces an added complexity for providing the requisite statistics for estimation. Another related work is [2], which discusses methods for efficiently estimating properties of joins. However, these methods only apply to primary key-foreign key joins, and cannot thus be applied to our estimation problem, which involves a more general kind of joins.

Another related line of works deals with rank-aware join algorithms (e.g. [11], [15]), that efficiently compute top-k over joins. Our approach extends the scope of these frameworks, to include rank-aware aggregation. General rank-aware query processing systems have been extensively studied in the literature (e.g. [14], and its extensions [12], [18]). These works, however, do not discuss aggregation, and cannot thus be applied in the entity-package finder setting. A rank-aware query processing system capable of top-k query processing over joins and aggregation is proposed in [13], but the techniques it presents assume that joins occur before aggregation (e.g. as is typically the case in SQL queries). These semantics are not compatible with the entity package finder problem (where joins need to be performed on top of aggregated results), and the techniques proposed in this work cannot be efficiently applied to our setting.

Scheduling accesses in an informed, data-adaptive manner, for increased performance, has been investigated in [7], in the context of the Threshold Algorithm ([10]). An adaptation of such techniques to our setting is not practical, as it has a large runtime overhead, due to properties of our setting.

7. CONCLUSIONS

In this work, we introduced the class of entity package finder queries. We examined algorithms resulting from adaptations of previous work, and we proposed Interleaved, an efficient algorithm to process such queries, by devising early pruning and termination strategies, in the presence of joins and aggregations, that do not depend on any estimates. We demonstrate the efficiency and scalability of our approach analytically and by experiments, on both real and synthetic large-scale data.

8. REFERENCES

[1] Opencalais. http://www.opencalais.com. Retrieved on June 23, 2008.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD Conference*, pages 275–286, 1999.

[3] A. Angel, S. Chaudhuri, G. Das, and N. Koudas. Ranking objects based on relationships and fixed associations. Tech.report, 2008. Available at http://www.cs.toronto.edu/ albert/docs/acdk-edbt09.pdf.

[4] D. E. Appelt and D. Israel. Introduction to information extraction. In *IJCAI Tutorial*, 1999.

[5] N. Bansal, S. Guha, and N. Koudas. Ad-hoc aggregations of ranked lists in the presence of hierarchies. In *SIGMOD Conference*, 2008.

[6] N. Bansal and N. Koudas. Blogscope: A system for online analysis of high volume text streams. In *VLDB*, pages 1410–1413, 2007.

[7] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[8] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD Conference*, pages 1247–1250, 2008.

[9] K. Chakrabarti, V. Ganti, J. Han, and D. Xin. Ranking objects based on relationships. In *SIGMOD Conference*, pages 371–382, 2006.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[11] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

[12] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid, H. G. Elmongui, R. Shah, and J. S. Vitter. Adaptive rank-aware query optimization in relational databases. *ACM Trans. Database Syst.*, 31(4):1257–1304, 2006.

[13] C. Li, K. C.-C. Chang, and I. F. Ilyas. Supporting ad-hoc ranking aggregates. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 61–72, New York, NY, USA, 2006. ACM.

[14] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: Query algebra and optimization for relational top-k queries. In *SIGMOD Conference*, pages 131–142, 2005.

[15] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, pages 43–52, 2008.

[16] K. Schnaitter, J. Spiegel, and N. Polyzotis. Depth estimation for ranking query optimization. In *VLDB*, pages 902–913, 2007.

[17] A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[18] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *ICDE*, pages 896–905, 2007.