Timely Semantics: A Study of a Stream-based Ranking System for Entity Relationships

Lorenz Fischer[1], Roi Blanco[2], Peter Mika[2], and Abraham Bernstein[1]

[1] University of Zurich, Department of Informatics, Zurich, Switzerland
{lfischer,bernstein}@ifi.uzh.ch
[2] Yahoo Labs, London, United Kingdom
{roi,pmika}@yahoo-inc.com

Abstract. In recent years, search engines have started presenting semantically relevant entity information together with document search results. Entity ranking systems are used to compute recommendations for related entities that a user might also be interested to explore. Typically, this is done by ranking relationships between entities in a semantic knowledge graph using signals found in a data source as well as type annotations on the nodes and links of the graph. However, the process of producing these rankings can take a substantial amount of time. As a result, entity ranking systems typically lag behind real-world events and present relevant entities with outdated relationships to the search term or even outdated entities that should be replaced with more recent relations or entities.
This paper presents a study using a real-world stream-processing based implementation of an entity ranking system, to understand the effect of data timeliness on entity rankings. We describe the system and the data it processes in detail. Using a longitudinal case-study, we demonstrate (i) that low-latency, large-scale entity relationship ranking is feasible using moderate resources and (ii) that stream-based entity ranking improves the freshness of related entities while maintaining relevance.

1 Introduction

In the past years, one of the major developments in the evolution of search engines has been the move from serving only document results to providing entity-based experiences. In contrast to the document results that are crawled from the Web, these experiences are typically built on top of a knowledge base assembled by the search engine provider from various sources of general and domain knowledge. All three major US search engines (Bing, Google, and Yahoo) have developed features that make use of such a knowledge base, and in particular to provide large information boxes which, at the time of writing, appear on the rightmost column of the interface for all three search engines. In all three cases, the displays also provide recommendations for related entities that the user may also want to explore.

Knowledge bases are typically organized in the form of an entity-relationship graph with additional facts attached to the entities and relationships. While the

facts represented in the graph rarely change, the timeliness of relationships can be significantly impacted by real world events. For example, in the domain of entertainment, a movie release could drive significant interest towards the collaborations of actors, or news of an impending celebrity divorce may raise interest into a couple. Similarly, in the domain of sports, a game could drive searches toward the players that participated in certain actions during the game, etc. The features that are used for measuring the importance of these relationships thus also need to be reassessed as a result of these events.

Entity recommender systems typically work by exploiting query logs for predicting the relevance of a related entity, as query logs provide an accurate reflection of current interests. Traditionally, such logs are collected and processed using offline, distributed batch processing systems such as Hadoop MapReduce.[3] These systems are designed to handle large volumes of data but at the cost of significant processing latency. More recently, a new class of distributed systems based on stream processing have become available, opening up the potential for new or improved applications of semantic technologies.

In this work, we describe *Sundog*, a stream processing based implementation of an entity-recommender system and show that by exploiting the temporal nature of search log data, we are able to significantly improve the quality of recommendations compared to static models of relevance, in particular with respect to freshness. The architecture of Sundog is based on a system that has previously been presented at ISWC – Spark [2]. To understand the differences in technology, we provide a comparison to the architecture of the batch-processing based predecessor. We then describe a longitudinal study that evaluates the relevance and freshness of the results computed by the system over a number of consecutive days, using different window sizes and temporal lag in computing the model. We show the benefits of using increasing amounts of data and reducing the lag in processing, namely a relevance and *freshness* increase of over 24% with respect to approaches that use stale data, in the best case. We conclude by discussing improvements and other potential applications of our work.

2 Related Work

Our *Sundog* system is an entity ranking system facilitating semantic search through the application of supervised machine learning techniques to features extracted from query log data. Hence, this section succinctly reviews the related work on (i) semantic search & entity ranking and (ii) temporal information retrieval.

With the introduction of entity-based experiences such as infoboxes, direct answers and *active objects* [16], the disambiguation of query intent and search results have gained in importance, because in these applications mistakes in query interpretation are immediately obvious to the user. However, the semantic gap between the words in user query and the descriptions of entities in the entity-graph can be significant [18]. Entity ranking, or ad-hoc object retrieval is aimed

[3] http://hadoop.apache.org

at finding the most relevant entity related to the user's query, and it has been the focus of many recent studies [2, 14, 20, 19, 26]. Pound et al. provide a query classification of entity-related search queries and define evaluation metrics for the entity retrieval task [20]. This task has also been the focus of evaluations in TREC [1] and other venues such as the SemSearch challenges [3]. A variant of the ad-hoc object retrieval task is the recommendation of related entities, where the focus is on ranking the relationships between a query entity and other entities in the graph, see Kang et al. [14] and van Zwol et al. [26]. More recently, Blanco et al. [2] present their work on the *Spark* system, which is a continuation of the work of Kang et al.

Temporal aspects have gained traction in information retrieval (IR) over the last couple of years and have found applications in document ranking [7, 6, 9], query completion [22], query understanding [15, 8, 17], and recommender systems [24, 5, 21]. Shokouhi et al. [22] analyse temporal trends and also use forecasted frequencies to suggest candidates for auto completion in web search. Kulkarni et al. analyse different features to describe changes in query popularity over time, to understand the intent of queries [15]. In [7], Dai et al. use temporal characteristics of queries to improve ranking web results using machine learned models. They use temporal criteria for their page authority estimation algorithms in [6]. More specifically, they propose a temporal link-based ranking scheme, which also incorporates features from historical author activities. Dong et al. identify *breaking news queries* by training a learning to rank model with temporal features extracted from a page index such as the time stamp of when the page was created, last updated, or linked to [8]. Elsas et al. analyzed the temporal dynamics of content changes in order to rank documents for navigational queries [10]. More related to the topic of query intent analysis, Metzler et al. [17] propose to analyse query logs in order to find base queries that are normally qualified by a year, in order to improve search results for *implicit year qualified* queries. The work that is probably most closely related to our study is by Dong et al. [9]. The authors use realtime data of the micro-blogging website Twitter to extract recency information and train learning to rank models, which in turn are used to rank documents in web search. The recency information from Twitter was then successfully used to rank documents, which promotes documents that are both more fresh and more relevant.

3 System Description

The entity ranking system employed at Yahoo – Spark – is implemented as a batch-processing based pipeline. For this study, we present Sundog, which implements parts of the Spark pipeline using a distributed stream processing framework. A full system description of the production system is beyond the scope of this paper and we refer to [2, 27] for details. However, for the sake of reproducibility and to understand the various design decisions made when building the system for our experiments, it is necessary to have an understanding of Spark. For this reason, we are first going to introduce the most important parts

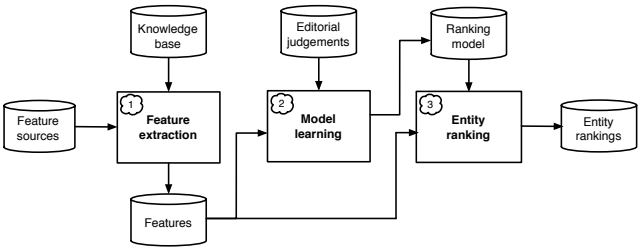

Fig. 1: High-level architecture of the Spark entity ranking system.

of the Spark processing pipeline before we elaborate in detail, how and in what aspects Sundog is different from the original system. We then describe the various performance optimizations we applied. We end the section with performance statistics of the system.

3.1 The Spark Processing Pipeline

Figure 1 gives a high level overview over the Spark ranking pipeline. The ranking essentially happens in three steps: Volatile data sources are used to generate co-occurrence features of entity pairs that are part of the relationships found in a semantic knowledge base (1). Data sources used for this step are Yahoo search logs, tweets from Twitter, and Flickr image tags. Note that for Sundog, we limited ourselves to only use search logs as input data. Next to features extracted from these volatile sources, semantic information such as the entity types and relationship types are leveraged as features. The next step involves the training of a decision tree model using editorial judgements that have previously been collected for a limited set of entity pairs (2). The resulting ranking model is then used to generate entity rankings for all the entity pairs for which features were extracted (3). Disambiguation is conducted in a post-processing step and it is based on a popularity measure derived from Wikipedia. For more information on pre- and post-processing as well as the serving facility, we refer to [2].

Model Learning & Ranking Spark employs learning to rank approaches in order to derive an efficient ranking function for entities related to a query entity.

Formally speaking, the goal of the Spark ranking system is to learn a function $h(\cdot)$ that generates a score for an input query q_i and an entity e_j that belongs to the set of entities related to the query $e_j \in \mathcal{E}^{q_i}$. Together, q_i and e_j are represented as a *feature vector* $\mathbf{w_{ij}}$ that contains one entry per feature extracted. The input of the learning process consists of *training data* of the form $\{T(q_i) = \{\mathbf{w_{ij}}, l_{ij}\}\}_{q_i \in \mathcal{Q}}$, where $l_{ij} \in L$ is a manually assigned label from a pre-defined set. Spark uses a 5-level label scale ($l \in \{$*Bad, Fair, Good, Perfect, Excellent*$\}$) and the assignment from examples (q_i, e_j) was done manually by professional editors, according to a pre-defined set of judging guidelines. The query set $\mathcal{Q}$ is comprised of editorially picked entities and random samples from query logs. This is expected to mimic the actual entity and query distribution of the live

system. The training set might also contain *preference* data, that is, labels that indicate that an entity is preferred over another one for a particular query. The ranking function has to satisfy the set of preferences as much as possible and at the same time is has to *match* the label in the sense that a particular loss function is minimized, for instance square loss $\frac{1}{|Q|}\sum_{q_i \in Q}\frac{1}{|E^{q_i}|}\sum_{e_j \in E^{q_i}}(l_{ij} - h(\mathbf{w_{ij}}))^2$, for a set of test examples.

Similarly to [27], Spark uses Stochastic Gradient Boosted Decision Trees (GBDT) for ranking entities to queries [12, 13]. GBRank is a variant of GBDT that is able to incorporate both label information and pairwise preference information into the learning process [25] and is the function of choice we adopted for ranking in Spark. The system was trained using ∼30K editorially labelled pairs and ten fold cross-validation. Each time a model is learned the system sweeps over a number of parameters (learning rate, number of trees and nodes, etc.) and decides on their final value by optimizing for *normalized discounted cumulative gain* (NDCG).

The features included in the system comprise a mixture of *co-occurrence features* and *graph-based features*. Co-occurrence features compute several statistics on mentions of pairs of entities appearing together in the data sources (conditional and joint probabilities, Kullback-Leibler divergence, mutual entropy, etc.). Other features include the types of entities and types of their relationships. In contrast to Spark, Sundog does not currently include graph-based features such as PageRank or the number of shared vertices (common neighbors) between two entities. It does, however, create features using various linear combinations of the features mentioned before as well as make use of the semantic features (type annotations). For a detailed description of these features we refer to [2].

3.2 The Sundog System

In this section we present the implementation details of the Sundog system. First, we describe the programming framework used to build the system. Next, we describe Sundog itself, before presenting a series of optimizations that were implemented to achieve the necessary performance.

Storm & Trident Sundog was implemented using the *Storm*[4] realtime computation framework. Storm is best described as the Hadoop MapReduce for stream processing. Similar to MapReduce, data is partitioned, distributed amongst, and processed by multiple compute nodes concurrently. A Storm application—a *topology*—is a directed graph consisting of spout and bolt nodes.

Trident is a higher level programming framework that is part of the Storm distribution. Trident offers higher level concepts such as aggregates, joins, merges, state queries, functions, filters, and methods for state handling. As the Sundog system relies heavily on the computation of state in the form of feature statistics that have to be computed continuously, we chose to use *Trident* for the implementation. In Trident, tuples are processed and accounted in mini-batches, offering consistency guarantees on a per-batch basis.

[4] http://storm-project.net

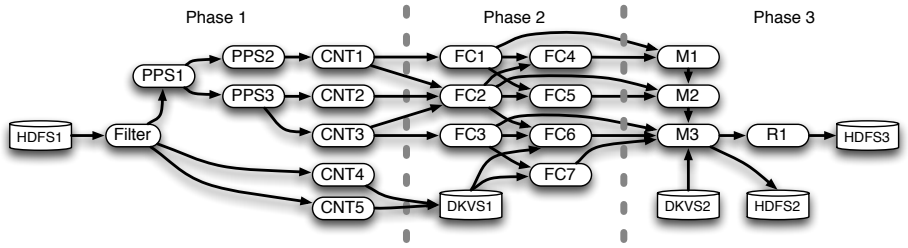

Fig. 2: The *Sundog* topology.

Topology Design The topology of Sundog can be roughly divided into three phases as depicted in Figure 2. Please note that for the sake of clarity, we show less nodes in the schematic depiction than the actual topology has. Each phase has to fully finish processing, before the next phase can start. For example, before we can compute the probability features of one mini-batch in phase 2, we first have compute the counter values for that mini-batch in phase 1. In the following paragraphs we are going to explain each of these phases in more detail.

In the first phase, query log data is read from the *Hadoop Distributed Filesystem* (HDFS). For the sake of reproducibility of our experiments and because we needed to be able to process historic log data, we chose to read from HDFS, rather than reading from a volatile message queue. We then filter out all search queries that do not contain at least one known entity name. The basis for this filtering is the same knowledge base (KB) that is used in the Spark system. For the experiments in this paper, we relied on the KB that was in production when we ran our experiments. From this reduced data stream, we already count certain events such as the number of distinct users or the number of search events. For other counters, we first have to build entity pairs from the query terms in a series of preprocessing steps (PPS1-3). We count the number of search events and unique users overall (for each entity pair and for each entity). As most of these counters count events and unique users per entity or entity pairs, the relevant data can readily be partitioned and the computation therefore run in parallel on multiple compute nodes. Some counters, however, have a global character and their value has to be aggregated across all data partitions. Their values are stored in an external distributed key-value store (DKVS1) to enable access from all compute tasks in later phases of the processing pipeline. The second phase consists of computing the actual feature values from the counter values (FC1-7). In the final phase (phase 3), the computed features are merged together and complemented with semi-static features that are read from a table in the distributed key-value store (DKVS2 in Figure 2). Semi-static features are features that do not change often, or not at all. For example the semantic type of an entity or a relationship between two entities is assumed to be mostly static. After all features have been merged together, a score is computed for each entity pair using a GBDT model. Both, the feature values and the final scores are written back to HDFS (HDFS2 and HDFS3 in Figure 2).

Optimizations A major difference between a Storm topology and a MapReduce-based data pipeline is that while between every MapReduce job data necessarily needs to be written to and read from disk. Storm does not have this requirement. Tuples can pass between bolt instances without ever being written to disk. As long as there is enough memory, state can be kept in memory for fast access. In its current implementation, Sundog only stores two counter values persistently in an external storage. All other information is kept in memory. For this reason, we implemented several optimizations that reduce the memory footprint and the network traffic incurred by the system. In this section, we list the most important ones.

HyperLogLog For some of the features, we need the number of unique users that searched for a given entity or a pair of entities. These counts are necessary to reduce the impact any single user can have on the ranking of an entity. For example, a fan of a football team may search for the name of his team very often together with the names of several other related entities. Normalizing with the number of unique users (rather than the number of search events) reduces the impact that any single search user has on the ranking.

The naïve way of counting uniques is to create a hash-set for each entity and entity pair – the values we want to count uniques for. For every search event, we could then add the user identifier to the hash-sets of the corresponding entity or entity pair. As hash-sets prevent duplicates from being stored, the size of the hash-set automatically represents the number of unique users that searched for a given entity or entity pair. The disadvantage of this method is, that we need one hash-set for every entity and entity pair and store the user identifiers of each user who searched for the given entity/entity pair. This has a worst case space complexity of $(e + ep) * u$, where e is the number of entities, ep is the number of entity pairs, and u is the number of users. With millions of users and millions of entity pairs, this number can become prohibitively large. To circumvent this, we used an implementation of the *HyperLogLog* algorithm proposed by Flajolet et al.[11]. More specifically, we used the *stream-lib*[5] library for approximate counting. The fact that approximate counting - or cardinality estimation - does not provide us with exact counts, should not have a great impact on our results, as we normalize all values with the same "imprecise" counts for unique users. For the experiments presented in this paper, we chose a relative standard deviation of 1% as the target accuracy for the *HyperLogLog* estimator.

Dictionary Encoding & Bloom Filters As we only have to work with a limited set of entity and relationship types, we use dictionaries to encode both of these values. The compressed dictionary file is so small, that we were able to include it in the jar file deployed on the servers to make it available on all machines.

As described in Section 3.2 the information about semi-static features is stored in a distributed key-value store. The data in this database is also used

[5] https://github.com/addthis/stream-lib

to filter invalid entity pairs early in the processing pipeline. This ensures that we do not compute feature values for entity pairs that eventually turn out to be invalid. For example the pair *"Brad Pitt - Zurich, Switzerland"* may be found in the search logs, because a user searched for these two entities in the same search query. However, the KB may not contain a relationship entry for the two entities *"Brad Pitt"* and *"Zurich, Switzerland"*. In fact, the vast majority of potential entity pairs are invalid as there are certain geographical locations that have names that (after the text normalization process) are lexicographically equivalent to some words in the English dictionary. For example, there are several villages in Norway with the name *"Å"*. This leads to many candidate entity pairs between "a" and other entities that have no semantic relationship with each other. In order to filter these, we need to check against the KB. To reduce the number of requests to the KB, and hence the number of network requests in the process, we use Bloom filters [4]. For the experiments presented in this paper, we added all entity relations in the KB into a Bloom filter. The resulting data structure is included in the application deployment.

3.3 Runtime Characteristics & Performance

There are many factors that influence the performance of a distributed system. For the sake of reproducibility, it may be of interest to the reader to learn more about the setup of our cluster and the configuration parameters that we used for the evaluation of Sundog. For this reason, we present the setup of our system in this section by first describing the general setup of the Storm cluster, before giving some insight into how we configured our topology in order to get better performance out of the hardware that we were able to use.

Cluster Configuration We ran our evaluations on a cluster of machines that are connected to a 1 Gbit/s network, each having 24 2.2GHz cores and 96GB of RAM. The service that starts and stops Storm worker instances is called the *supervisor* service. There is one supervisor per machine in the cluster. All supervisors are centrally managed by a master server, the *nimbus* node. Communication between the nimbus service and the supervisors happens over a *Zookeeper*[6] cluster. The *nimbus* server schedules work among the available supervisor nodes. For our experiments, we were given exclusive access to 40 machines. We configured *Storm* to start 8 worker instances (JVMs) per supervisor, each having 12GB of RAM.

Job Configuration & Performance Table 1 lists setup parameters we used to configure Storm and Trident: We ran our experiments on 40 supervisors, each having 8 workers which resulted in 320 workers in total. These workers were executing 2449 task instances, of which 40 were spout instances and 2087 were regular bolt instances. The remaining bolt instances are acker-instances or Trident coordinator bolts. We ran one spout instance for each machine. Each of

[6] http://zookeeper.apache.org

these read and emitted 100,000 log lines per batch. One batch took on average 40 seconds to complete, which means that the system ingested about 100,000 log lines per second. We found that neither increasing nor decreasing the batch-size led to increased throughput. Most likely a result of the bookkeeping overhead of Storm becoming proportionally more expensive with smaller batches, while larger batches just increased the processing time per batch. The system transferred about about 2.5 million messages per second within the topology. As running multiple batches concurrently did not yield higher throughput and only increased the chances for batches to time out, we always only processed one batch at a time. This led to the situation that we barely used all of the available resources, because, as mentioned in Section 3.2, certain parts of the computation need to wait for other parts to complete. This suggests that there may be further potential improvements in terms of resource utilization.

Parameter	**Value**
Workers	320
Spouts (Spout Tasks)	1 (40)
Bolts (Bolt Tasks)	30 (2087)
Total Task Instances	2449
Concurrent Batches	1
Batch Size (log lines)	100'000
Average Batch Time (Seconds)	≈ 40
Log Lines per Second	100'000
Transferred Messages per Second	2.5 mio

Table 1: Sundog configuration parameters and performance numbers of a typical evaluation run.

While the underlying platforms of Sundog and Spark are vastly different and the performance indicators can therefore not easily be compared directly, it is interesting to note, that even though Spark is running on a cluster that has two orders of magnitude more machines, Sundog is still able to process comparable amounts of data in about $\frac{3}{4}$ of the time used by Spark.

4 Evaluation

Sundog allows us to compute feature values and entity rankings in much less time compared to the old Spark system. This in turns enables us to use more recently collected data for the ranking process. Hence, we are interested in three things: First, we investigate the impact of data recency on the entity rankings. For this we are interested in measuring the quality of the rankings in terms of freshness and relevance. Secondly, we analyze if fresh rankings are more useful to users. Lastly, we evaluate how the amount and the age of data used to train the system impact performance.

4.1 Experimental Setup

We evaluated the rankings that Sundog produces on four different days over the course of a week. We had human editors assess the rankings with regards to relevance and freshness on each day. In this section, we present the experimental setup of our evaluation. First, we describe the data sets that we collected, before we describe in detail how our editors assessed the generated rankings.

The Data For our experiments, we produced entity rankings using search log data of three time periods. For each time period we grouped the log files in sets of different sizes (windows). Each log file set $s_i \in S$ has a window of size w_i and an end date d_i. The window size is inclusive. Hence, for a set s_i with a window size $w_i = 7$ and end date $d_i = 2014/01/12$, the respective start date is defined as $d_i - w_i + 1 = 2014/01/06$. We differentiate between three different time periods or epochs, so three collections of *old*, *recent*, and *new* sets of log files. As we ran our experiments on 4 different days, the values for the *new* epoch changed. The end date for the *new* epoch is defined as:

$$d_n \in \{2014/01/20, 2014/01/21, 2014/01/22, 2014/01/23\}$$

For the *recent* and the *old* epoch we chose $d_r = 2014/01/12$ and $d_o = 2013/12/31$, respectively, to simulate the situation in which the rankings would be used during a period of two to three weeks. For each period we compiled a data set of three different sizes $w \in \{1, 7, 30\}$. Table 2 lists the resulting data sets.

Epoch	Window	Dates
Old	1 Day	2013/12/31
	7 Day	2013/12/25 – 2013/12/31
	30 Days	2013/12/2 – 2013/12/31
Recent	1 Day	2014/1/12
	7 Day	2014/1/6 – 2014/1/12
	30 Days	2013/12/14 – 2014/1/12
New	1 Day	2014/1/20...23
	7 Day	2014/1/14...17 – 2014/1/20...23
	30 Days	2013/12/22...25 – 2014/1/20...23

Table 2: Data sets collected for the evaluation.

For each of the data sets we first had Sundog generate the feature values which are stored in files. We then used these feature files to train *Gradient Boosted Decision Tree* (GBDT) models (see 3.1 for details). The resulting models were then used to generate the entity rankings, again stored in files. For each feature file and its corresponding model, we generated one ranking file. In addition, to test the performance of a model that has been generated with an old feature file on freshly generated feature values, we also generated some ranking files using models trained on old data and feature files extracted from new search log data. Note that for all rankings where we used models trained with historic data, we only scored the feature files on models of the corresponding window

size, as the feature values in the feature files would otherwise be incompatible with the models.

The resulting ranking files contained a ranking score for each entity pair that at least one user searched for within the corresponding time window. As the number of such rankings can be quite large, we restricted ourselves to evaluate only a subset of all pairs. As we are mostly interested in evaluating for freshness, we selected the top-60 of all queries that matched the label of entities in the KB. This ensured, (i) that we only select queries for which related entities would actually be shown on the search page, and, as the entities in question were "trending", (ii) increased the likelihood that recency would be a factor for the relationships. We then took all entity pairs that we could find from all 15 ranking files for that day and pooled the query-entity pairs. With at most 10 related entities on the result page, this yielded a pool of at most $60 \times 15 = 900$ entity pairs per day - which was the upper limit of entity pairs that our human editors could evaluate in respect to relevance and freshness in a day. Table 4 lists the exact numbers of query-pairs for each day.

Editorial Judgement We asked a group of expert search editors working for Yahoo to judge entity pairs in terms of relevance and freshness. Table 3 lists the categories from which the editors could select. The editors were trained and instructed to judge each relationship from the viewpoint of "today". We asked the editors to research the relationships which they did not know about, in order to provide a well founded judgment.

Recency Categories		Relevancy Categories	
Super Recent	Is current today or yesterday	Super Related	Most interesting factual relationship
Very Recent	Was current the past week	Closely Related	Related in a meaningful or useful way
Recent	Relevant in the past year	Mostly Related	A little off, but makes sense
Reasonable	A bit old, but still popular	Somewhat Related	Not a meaningful or useful suggestion
Outdated	There are better connections	Embarrassing	Does not make sense
NA or NJ	Freshness is not a factor	N/J	No judgement possible

Table 3: Available recency and relevance categories and their description.

Date	Pairs	Super-Recent	Super-Related
2014/01/20	819	57	290
2014/01/21	696	34	184
2014/01/22	865	54	171
2014/01/23	785	34	196

Table 4: The number of query pairs evaluated on each day with the corresponding number of pairs that were judged *Super-Recent* or *Super-Related*, respectively.

We measure the performance on both *relevance* and *freshness* using standard metrics such as normalized discounted cumulative gain (NDCG), precision, and

mean average precision (MAP). Given that we are considering freshness as a discrete, graded variable we report on the same metrics as relevance, but using the editorial labels for recency.

4.2 Results & Discussion

Fresh Data Is Better In Figure 3 we present our findings on the impact of data freshness on relevance and freshness scores: In the two charts in the upper row we plotted the NDCG scores using the top-10 and the top-5 results. We chose top-10, because Spark always shows the top-10 ranked related entities. Sundog may not be able to always find 10 related entities. This has several reasons: Firstly, Sundog only uses one of the four data sources that Spark uses. Secondly, while Spark uses default values for all features and entity pairs that could not be found in the data, Sundog only computes features values, and hence rankings, for entity pairs that we were able to find in the data. As we are currently mostly interested in freshness of relationships it makes sense not to include relationships that were not of any importance to our users during the time we collected the data. For this reason, Figure 3 also shows the NDCG values for the top-5 ranked entities.

It is apparent, that for sufficiently large time windows, *new* data always produces entity-rankings that are both fresher and also more relevant in general. If the data only contains log data from a single day, we see that while the data that was collected most recently still consistently produces superior rankings, the difference between the rankings of the *recent* data compared to the *old* data is negative. Looking at the numbers in Table 5a we can see a similar picture: Using window sizes of 7 and more days, we observe a significant improvement in terms of freshness when using more recent data.

	Old	Recent	New
1	0.3600	0.3446 (-4.20%)	0.3868^{*} (+11.13%)
7	0.3945	0.4317^{*} (+9.44%)	$0.4870^{*\dagger}$ (+24.38%)
30	0.4569	0.4994^{*} (+9.30%)	$0.5335^{*\dagger}$ (+16.75%)

(a) Freshness

	Old	Recent	New
1	0.4499	0.4522 (+0.66%)	$0.4958^{\dagger}$ (+10.20%)
7	0.5107	0.5913^{*} (+15.80%)	0.6123^{*} (+19.90%)
30	0.6041	0.6588^{*} (+7.71%)	0.6589^{*} (+9.05%)

(b) Relevance

Table 5: NDCG-10 improvements reported over *old* baseline. * indicates a significant improvement over *old*, and † over *recent* (p-value < 0.05, paired two-sided t-test). Values for *new* are averaged over all four days.

In the graphs in the lower row of Figure 3, we compare the relevance and freshness measured using several metrics such as precision (P), MAP, and NDCG

using a cutoff of 5 and 10, respectively. These charts confirm that our hypothesis also holds for this analysis: Rankings produced from *new* data score higher than rankings produced from historic data.

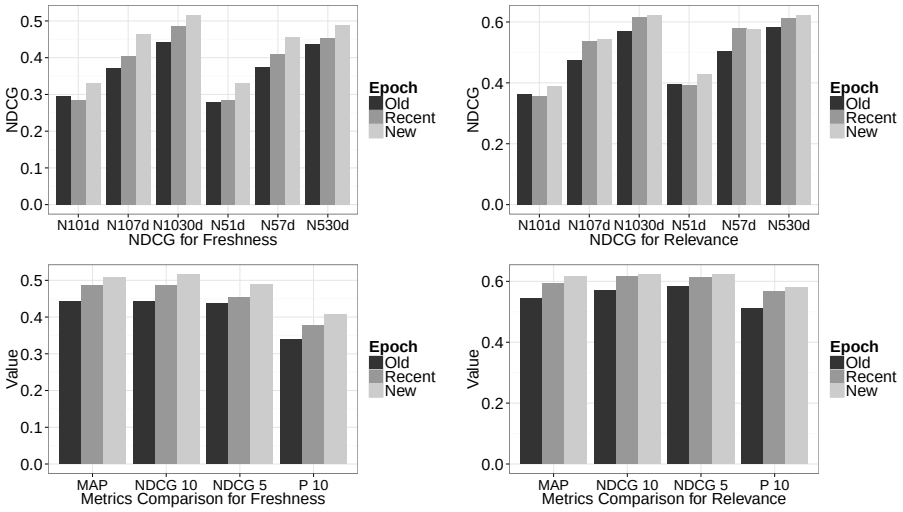

Fig. 3: Top: NDCG for the top 10/5 ranked entities for freshness and relevance Bottom: Comparing several metrics for freshness and relevance for a 30 day window

	$p(NotRel.\|\cdot)$	$p(Rel.\|\cdot)$
Super Recent	0.15	0.85
Very Recent	0.21	0.79
Recent	0.50	0.50
Reasonable	0.31	0.69
Outdated	0.57	0.43
NA or NJ	0.45	0.55

(a) Relevance

	$p(\cdot\|NotRel.)$	$p(\cdot\|Rel.)$
Super Recent	0.02	0.11
Very Recent	0.05	0.14
Recent	0.23	0.17
Reasonable	0.14	0.25
Outdated	0.52	0.41
NA or NJ	0.05	0.03

(b) Freshness

Table 6: Distribution of recency across freshness and relevance values. "Super Related" and "Closely Related" collapsed into "Relevant" (Rel./Not Rel.).

Fresh is Relevant Table 6a shows the probabilities of different freshness labels conditioned to observing relevance and non-relevance, respectively. Relevance labels have been collapsed in the table, this is, we deemed the labels *Super Related* and *Closely Related* as relevant and all other labels as not-relevant. The results suggest that freshness is a good indicator for relevance for the *Super Recent* and *Very Recent* categories. On the other hand, looking at Table 6b, we observe that relevance is not a good indicator for freshness. Intuitively this makes sense as it seems logical to assume that there are many more relationships between entities that may, although being relevant, not be of immediate importance in terms of recency. Overall, Pearson's correlation coefficient between labels is 0.28, which indicates that there is only a slight correlation between them.

More Data Is Better In all but one cases, having more data available to generate the rankings resulted in better performance in terms of relevance (top-right in Figure 3). Looking at the freshness evaluation, we observe a similar behavior (top-left in Figure 3) with the exception of the NDCG values computed using *new* data for the 7-day window, that for both, the top-10 and the top-5 ranks scored higher than the corresponding NDCG values for the 30-day window computed using "old" data. While this observation is consistent with the machine learning literature, it also shows that data that is more fresh can compensate in situations where only very little historic data can be collected.

Performant Recent Models In order to asses how well the GDBT models we employed generalize for unknown data, we used models of varying age to rank feature data generated from the most recent log data. The results of this comparison are shown in Figure 4: Using 20 day old data to train the models (Model Epoch = Old) yields worst performance for both freshness and relevance for all time windows, which suggests that the age of a trained model has an impact on performance. Comparing the performance achieved when using 10 day old (Model Epoch = recent) and current (Model Epoch = new) data, however, we can see that freshly trained models do not necessarily deliver better performance. This suggests, that while fresh data is important for ranking entities, the training of models is less time critical.

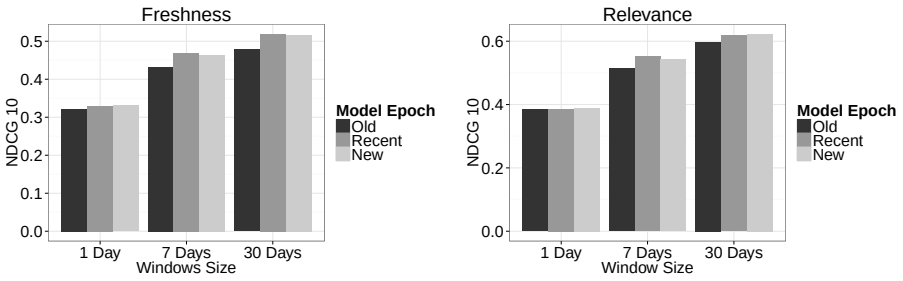

Fig. 4: Comparing the effect of the model age on freshness (left) and relevance (right). NDCG values obtained by applying new feature values on models of varying age.

5 Conclusions & Future Work

We presented an evaluation of Sundog, a system for ranking relationships between entities on the web using a stream processing framework. Sundog is able to ingest large quantities of data at high rates (orders of magnitude more than a legacy batch-based system) and thus allows for adapting ranking into a live setting, where the ordering of elements (entities) displayed to the user might change with small time delays. This can be accomplished by inspecting relevance signals coming from query logs and updating feature values on the fly.

This architecture enabled us to investigate the tradeoff between data recency and relevance in a live setting, where rankings can change every day. We ran live experiments on four different days using real queries, generating rankings that were evaluated by professional human editors with regards to their *relevance* and *recency*. We trained different models, using old, recent and new data and reported their performance. It is apparent, that for sufficiently large time windows, *new* data always produces entity-rankings that are both, more fresh and also more relevant, with improvements reaching up to 24% in NDCG. We observed, that recency of input data can even compensate for reduced amounts of data, which is traditionally thought of as being a primary factor for the performance of machine learning models. Additionally, the ranking models we deployed are robust enough to be able to generalize well 10 days after they have been trained, even when feature values for query-entity pairs have changed over time. This suggests that while being able to process recent data is crucial, realtime re-learning does not impact performance as much (if at all).

While the source code of the system as well as the search log data used in the study are proprietary to Yahoo or cannot be released to the public for privacy reasons, we do provide a detailed description of the system and the data, which does allow for reproducibility of our results. For example, similar data sets that could be used are tweets from Twitter or image tags from Flickr. In addition, Sundog is built using open source software, e.g. Apache Storm.

In future work, we will explore adaptions to the ranking model in more depth and also investigate, how freshness and relevance can be combined into one objective function. Currently, the models are learned by trying to maximize the relevance score. While recency and relevance are not necessarily two orthogonal performance characteristics, they can differ. The way in which one could combine these two aspects is not clear, yet. This, as well as an investigation of techniques with which recency and relevance can be independently measured without trained editors, remains future work. Additionally, we are interested in equipping the system with an online learner in order to make use of user feedback information (clicks) in real time. Finally, additional work on recency features is also necessary in order for the ranking models to be able to capture time dependent characteristics as for example concept shifts [23].

Acknowledgments We would like to thank B. Barla Cambazoglu, Alice Swanberg and her team, Derek Dagit, Dheeraj Kapur, Bobby Evans, Balaji Narayanan, and Karri Reddy for their invaluable support.

References

1. Balog, K., Serdyukov, P., de Vries, A.P.: Overview of the trec 2011 entity track. In: Voorhees, E.M., Buckland, L.P. (eds.) TREC. National Institute of Standards and Technology (NIST) (2011)
2. Blanco, R., Cambazoglu, B.B., Mika, P., Torzec, N.: Entity Recommendations in Web Search. In: ISWC 2013 (2013)
3. Blanco, R., Halpin, H., Herzig, D.M., Mika, P., Pound, J., Thompson, H.S., Tran, T.: Repeatable and reliable semantic search evaluation. J. Web Sem. 21 (2013)

4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7) (1970)
5. Chen, W., Hsu, W., Lee, M.L.: Modeling user's receptiveness over time for recommendation. SIGIR (2013)
6. Dai, N., Davison, B.D.: Freshness matters: in flowers, food, and web authority. In: SIGIR 2010 (2010)
7. Dai, N., Shokouhi, M., Davison, B.D.: Learning to rank for freshness and relevance (2011)
8. Dong, A., Chang, Y., Zheng, Z., Mishne, G., Bai, J., Zhang, R., Buchner, K., Liao, C., Diaz, F.: Towards Recency Ranking in Web Search. In: WSDM (2010)
9. Dong, A., Zhang, R., Kolari, P., Bai, J., Dias, F., Chang, Y., Zheng, Z.: Time is of the essence: improving recency ranking using twitter data. In: WWW (2010)
10. Elsas, J.L., Dumais, S.T.: Leveraging temporal dynamics of document content in relevance ranking. WSDM (2010)
11. Flajolet, P., Fusy, E., Gandouet, O., Meunier, F.: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. DMTCS Proceedings (2008)
12. Friedman, J.H.: Stochastic gradient boosting. Computational Statistics & Data Analysis (1999)
13. Friedman, J.: Greedy function approximation: a gradient boosting machine. Annals of Statistics (2001)
14. Kang, C., Vadrevu, S., Zhang, R., Zwol, R.V., Pueyo, L.G., Torzec, N., He, J., Chang, Y.: Ranking related entities for web search queries (2011)
15. Kulkarni, A., Teevan, J., Svore, K.M., Dumais, S.T.: Understanding temporal query dynamics. In: WSDM. ACM Press (2011)
16. Lin, T., Pantel, P., Gamon, M., Kannan, A., Fuxman, A.: Active objects: actions for entity-centric search. In: WWW (2012)
17. Metzler, D., Jones, R., Peng, F., Zhang, R.: Improving search relevance for implicitly temporal queries. In: SIGIR. No. 1, ACM Press (2009)
18. Mika, P., Meij, E., Zaragoza, H.: Investigating the Semantic Gap through Query Log Analysis. In: ISWC (2009)
19. Pehcevski, J., Thom, J.A., Vercoustre, A.M., Naumovski, V.: Entity ranking in Wikipedia: utilising categories, links and topic difficulty prediction. Information Retrieval 13(5) (Jan 2010)
20. Pound, J., Mika, P., Zaragoza, H.: Ad-hoc object retrieval in the web of data. WWW (2010)
21. Ren, Z., Liang, S., Meij, E., de Rijke, M.: Personalized time-aware tweets summarization. SIGIR (c) (2013)
22. Shokouhi, M., Radinsky, K.: Time-sensitive query auto-completion. In: SIGIR. ACM Press (2012)
23. Vorburger, P., Bernstein, A.: Entropy-based concept shift detection. In: ICDM. IEEE Computer Society (2006)
24. Yuan, Q., Cong, G., Ma, Z., Sun, A., Thalmann, N.M.: Time-aware point-of-interest recommendation. SIGIR (2013)
25. Zheng, Z., Zha, H., Zhang, T., Chapelle, O., Chen, K., Sun, G.: A general boosting method and its application to learning ranking functions for web search. In: Neural Information Processing Systems (2008)
26. van Zwol, R., Garcia Pueyo, L., Muralidharan, M., Sigurbjörnsson, B.: Machine learned ranking of entity facets. In: SIGIR. ACM Press (2010)
27. van Zwol, R., Pueyo, L.G., Muralidharan, M., Sigurbjornsson, B.: Ranking Entity Facets Based on User Click Feedback. In: ICSC. Ieee (Sep 2010)