Fast, Expressive Top-k Matching *

William Culhane
Purdue University
West Lafayette, IN, USA
wculhane@cs.purdue.edu

K. R. Jayaram
IBM Research
Yorktown Heights, NY, USA
jayaramkr@us.ibm.com

Patrick Eugster
Purdue University
West Lafayette, IN, USA
peugster@cs.purdue.edu

ABSTRACT

Top-k matching is a fundamental problem underlying online advertising platforms, mobile social networks, etc. Distributed processes (e.g., advertisers) specify predicates, which we call *subscriptions*, for *events* (e.g., user actions) they wish to react to. Subscriptions define *weights* for elementary *constraints* on individual event *attributes* and do not require that events match all *constraints*. An event is multicast only to the processes with the k highest *match scores* for that event – this score is the aggregation of the *weights* of all constraints in a subscription matching the event.

However, state-of-the-art approaches to top-k matching support only rigid models of events and subscriptions, which leads to suboptimal matches. We present a novel model of weighted top-k matching which is more expressive than the state-of-the-art, and a corresponding efficient algorithm. Our model supports attributes with *intervals*, weights specified by producers of events *or* by subscriptions, *negative* weights, *prorating* of matched constraints, and the ability to *vary scores dynamically* with system parameters. Our algorithm exhibits time and space complexities which are competitive with state-of-the-art algorithms regardless of our added expressiveness – $O(M \log N + S \log k)$ and $O(M N + k)$ respectively, with N the number of constraints, M the number of event attributes, and S the number of matching constraints.

Through empirical evaluation with both statistically generated and real-world data we demonstrate that our algorithm is (a) equally or more efficient and scalable than the state-of-the art without exploiting our added expressiveness, and it (b) significantly outperforms existing approaches upgraded – if possible at all – to match our expressiveness.

1. INTRODUCTION

Top-k matching is an important computation problem requiring fast execution [14, 17]. In top-k matching, entities wishing to be matched, e.g. advertisers, request to add subscriptions to a system, where each subscription corresponds to an advertisement ("ad"). When an event, such as a browser event representing a webpage view, triggers the matching algorithm, the algorithm finds and returns the k best matching subscriptions based on the user's interests and attributes (e.g., sex, age, location, etc.). Corresponding middleware solutions are widely used for online ads, e.g., text or banner ads based on user profiles and browsing histories, streaming videos ads, social network "online deals", etc.

1.1 Expressive Matching

Targeted ads may affect purchasing intent by 65% [12] and click through rates by 670% [21], so *effective* matching is vital to advertisers. As online behavior and personal information are more effectively tracked, especially via social media, using those data for matching becomes more relevant. In 2011 93% of companies reported using social media for marketing [19], and not too long ago Facebook announced a program to serve demographically targeted video ads to its users [15]. Users of social media are however quickly deterred by unfocused ads [16]. This can be avoided by *more expressive* events, subscriptions, and matching between these.

Consider *ad exchanges* – marketplaces for ads used by Google, Facebook, Yahoo!, and others. Ad exchanges can be viewed as networks of processes routing advertisers to consumers (email users, website visitors, video viewers, etc.). A consumer's arrival can be modelled as an event with attribute values determined by scripts (read from cookies or otherwise inferred). The values of such attributes representing user information are typically actual values. Some attributes may instead have "*undefined*" values or be captured by *intervals* of values. An example event for a user arrival is `{fName: Jack, lName: UNKNOWN, age: [18..29], state: Indiana}`.

Advertisers typically do not need individual constraints to match a consumer entirely. For example, an advertiser for spring break airfares from Chicago to Cancun submits a subscription with preferences for consumers of the ad, e.g., `(age` $\in$ `[18,24]` $\wedge$ `state` $\in$ `{Indiana, Illinois, Wisconsin})`. Advertisers assign a *weight* to each attribute and request the ad be delivered to consumers with a significant combined weight of matching attributes. The weight on the attribute signifies the relevance of serving the ad to the consumer; the advertiser assigns higher weights to more relevant attributes.

Campaigns have fixed budgets to spend in a specified time period, so there is an advantage to *dynamically adapt the scoring mechanism* to fully utilize budgets without overspending. Consider an ad campaign for a concert with a fixed budget for a given period of time. The campaign does not want to exhaust the budget within a day nor serve so

*Financially supported by US NSF grants 0644013 and 0834529 and Purdue Research Foundation grant 205434.

Middleware '14, December 08 - 12 2014, Bordeaux, France

ACM 978-1-4503-2785-5/14/12 $15.00
http://dx.doi.org/10.1145/2663165.2663326.

few ads as not to use it entirely. Rather than force the campaign's owner to monitor progress and change the weights in an attempt to match at the desired rate, the system can track the budget, the length of the campaign, the desired rate of matching, and the historical rate of matching to *adjust a score multiplier* to achieve the desired matching rate.

For a consumption medium, a number of ads per access should be chosen which is minimally intrusive and economically viable. Each access generates an event with the available known attributes and sends it to the exchange. The exchange returns the k ads best suited to display for while balancing the advertiser preferences as well.

1.2 Fast *and* Expressive Top-k Matching

One key bottleneck in the design of middleware systems for top-k matching is the latency of the matching algorithm, i.e, time taken to compute matches for events. Several specialized approaches have been developed for top-k matching. Examples are Fagin's seminal aggregation algorithm [9], SOPT-R Trees and scored segment and interval trees in combination with Fagin's algorithm [14], and BE*-Trees [17].[1] For scalability, a distributed overlay can be used (e.g., [4, 7]). *Distributed* top-k matching aggregates and prunes sets from top-k matching at individual nodes, so any expressiveness desired for the system must be implemented centrally.

Motivated by recent application scenarios, we propose in this paper a more *expressive* model of weighted partial top-k matching than considered in prior art, and an *efficient*, novel algorithm dubbed Fast eXpressive Top-k Matching (FX-TM) for implementing it. Our model supports:

(a) explicit handling of *interval attributes* and optional *prorated scoring* when there is partial overlap between subscriptions and events. This corresponds to scenarios where exact values for user attributes are not known (e.g., targeted age [18,24] and consumer age [20, 30]).

(b) attribute weight on *either* the subscription *or* the event used during aggregation. For instance, a company may favor experience over applicant location while a job seeker may prefer proximity over experience requirements. Our model allows each of these, and can switch between approaches for each matching iteration.

(c) *non-monotonic aggregation functions* and *mixed positive and negative weights* forbidden by some of the state-of-the-art. This is useful when some attribute values are undesirable, e.g. an age range below the voting age, while others, e.g. income, are desirable.

(d) missing attributes and wildcards, finding matches with the best scores even if a match is not *exact* (i.e., does not match all attributes). This is useful as some groups are more prone to self disclosure [5]. Available data is matched; missing data does not disqualify a match.

(e) *dynamic alteration of scores* based upon the rate a subscription is matched to events and versus its specified preferred rate. This helps an ad campaign match as many events as its budget allows over a specified period of time without depleting its budget early.

Our output-sensitive algorithm runs a single match in time $O(M \log N + S \log k)$ for N subscriptions, M attributes in the event, and S matching elementary constraints. The complexities hold regardless of chosen expressiveness options.

[1]Note that Fagin has been recently awarded the Gödel prize for his seminal work on top-k matching.

1.3 Contributions and Roadmap

After surveying prior art (Section 2), this paper contributes:

1. a model of expressive weighted top-k matching with the features outlined above (Section 3);
2. an efficient algorithm FX-TM of our model, including a score adjustment device to achieve desired matching rates for each subscription (Section 4);
3. a formal complexity analysis of our algorithm qualifying its scalability (Section 5).
4. an implementation of our FX-TM algorithm including its distributed application over several nodes with subsequent aggregation (Section 6);
5. an empirical evaluation consisting of statistical micro-benchmarks and two benchmarks derived from real-life data. We show that FX-TM performs at least as well as state-of-the-art solutions before considering our added expressiveness and dynamic score adjustment; when considering these, FX-TM surpasses existing approaches after attempting to upgrade those (Section 7).

Section 8 concludes with an outlook on future research.

2. EXISTING WORK

This section presents prior art and its limitations.

2.1 Top-k Matching

Fagin proposes the seminal algorithm [9] for top-k matching. It runs on databases, so querying sorted lists for individual attributes is easy, and it uses these sorted lists and an aggregation function to create an aggregate order. Matching takes $O(N^{(M-1)/M}\, k^{1/M})$ time *starting from the point the sorted lists are available* – N is the number of objects in the system (our subscriptions) and M the number of attributes.

Fagin and Wimmers [11] expand this to incorporate weights that can vary per attribute without affecting the running time. In line with the database querying model, these weights are determined by the query (corresponding to our events) rather than the objects in the database (our subscriptions).

Machanavajjhala et al. [14] at Yahoo! attempt the first top-k system without a database for the specific use of targeted ads. Their work shows such an approach can efficiently find the attribute level results when attributes can be stored in sorted order, and those results used in Fagin's algorithm.

Sadoghi and Jacobsen [17] also approach the problem from a storage perspective instead of focusing primarily on aggregation. Their BE* tree uses an alternating clustering and dimension partitioning strategy. They choose the partitioning such that the most divergent dimensions are used first to prune the search space quickly. They show that this approach is experimentally competitive to a multi-dimensional storage tree on generated and real-world data.

Recent research into top-k matching focusses on performing rankings with uncertain data. Dylla et al. [8] and Song et al. [18] consider approaches to display the k best suspected matches when some of the values in the data are unknown or known only with some probability. Both methods require some additional overhead to handle this extra requirement.

Search engines and ad companies have their own approaches, but these are proprietary and unavailable to the public.

2.2 Distributed Top-k Matching

Several researchers consider *distributing* the top-k matching problem. Most proposals rely on running a matching algorithm on nodes on *partitions* of the data, then aggregating the partial results in some way. Improving the core top-k matching algorithm thus improves the distributed system.

Fagin mentions that the data can be housed on multiple systems, but does require all of the accesses to be done from a single computer to run the algorithm. Cao and Wang [4] distribute Fagin's algorithm by running it at multiple nodes, and then combining the partial results at a single node. The process involves running the algorithm and pruning before sending the partial results to a designated node for merging. The paper focuses on the best way to do this while conserving bandwidth. Fagin proposes a similar threshold algorithm [10]; that paper includes extensive proofs on how well similar algorithms perform under various scenarios.

These approaches rely on in part on Fagin's algorithm. Machanavajjhala et al. [14] theorize about an alternative possibility of using a publish/subscribe system to find matches, presumably similar to what is described by Aguilera et al.[1] with more robust matching, but claim it is inefficient and do not implement it. What they do implement uses Fagin's algorithm. Thus any improvements to a local algorithm are immediately pertinent to distributed top-k matching.

2.3 Limitations

Efforts based on Fagin's work exhibit several limitations. Among them is assuming the time for ordered access on attribute matching can be ignored. With proration and dynamic multipliers, scores cannot be known ahead of time, so subscriptions cannot be stored in sorted order, and sorting is run during retrieval. This is required for each of attribute, which takes $O(MS\,logS)$ time for S matching subscriptions per M event attributes prior to running the algorithm.

One requirement of Fagin's algorithm propagated to other work is the explicit need for *monotonicity* in the scoring mechanism [9]: component scores from attributes are considered sequentially, and the aggregate score is required to either exlusively increase or decrease (or stay the same) as the algorithm runs. This works well for some functions, e.g. minimum and maximum subscore. However, even an aggregation function as simple as summation is not monotonic when considering the possibility of positive *and* negative weights. For example, consider aggregating the score over a match with component scores {.2, .2, -.1}. After the initial component, the score is .2. With the next component it *increases* to .4 and then *decreases* to .3. As long as two component scores aside from the first have opposite signs, monotonicity is not guaranteed. Consider a political campaign trying to serve ads. The campaign would prefer to serve ads to users with some attributes (gender, income, etc.). Those below the voting age should be avoided, corresponding to a negative weight for this campaign.

Fagin and Wimmers [11] present a device for weighting attributes in a match, but weighting can only be on what we call an event. In the case of advertising, the ad consumer has a smaller stake in the match than the advertiser. Thus it makes more sense to allow the advertisers to specify weights independent of each other as they will value different attributes for targeting demographics. Advertisers are the objects in the system, the subscriptions, and the ad serving is a result of the events; this example of weighting cannot be done with events as proposed by Fagin and Wimmers.

BE* trees [17] are easily modified to overcome these limitations despite being best tuned to events with attributes containing single values. However, as we will demonstrate empirically, BE* trees are not always the fastest approach.

None of this research considers dynamic score adjustment based on system behavior.

3. MODEL

This section defines events, subscriptions, and matching.

3.1 Matching Events and Subscriptions

We consider an event e to be a set of attribute/interval pairs $\{a_1 : [v_1, v_1'], \ldots, a_l : [v_l, v_l']\}$. Events are only required to include attributes whose values are known, but may specify attributes with `UNKNOWN`. For simplifying presentation and comparison with predating work, we consider subscriptions to be *conjunctions* of constraints.

A subscription is thus in the following represented as a predicate ϕ based on the following grammar (extended BNF):

$$\begin{array}{lrcl} \textit{Predicate} & \phi & := & \phi \wedge \delta \mid \delta \\ \textit{Constraint} & \delta & := & a \in [v, v'] : w \end{array}$$

Interval subscriptions are predicates where the only boolean operator is $\in$. We call to subscriptions where predicates have relational operators *regular* subscriptions. A predicate `x>100` for some integer `x` is expressed as `x`$\in$ [`101`, `MAX_INT`]. A predicate for a single value or member of a set is represented by $v = v'$ so the "interval" contains only a single value. Each Constraint has an optional weight w attached.

To support weights in events instead of subscriptions, one can consider the events being augmented with weights w on attributes, $\{\langle a_1 : [v_1, v_1'], w_1\rangle, \ldots, \langle a_l : [v_l, v_l'], w_l\rangle\}$, which, when they exist, override the weights in subscriptions.

Interval subscriptions are as expressive as regular subscriptions. Matching involves substitution, i.e., substituting the values of attributes in events into constraints in a subscription. We denote this for a single constraint by $\delta(e)$. For example, if $e = \{a_1 : [v_1, v_1'], \ldots, a_l : [v_l, v_l']\}$, and $\delta = a \in [v_1'', v_1''']$, $\delta(e) = [v_1, v_1'] \in [v_1'', v_1''']$. $\delta(e)$ evaluates to TRUE or FALSE. When an attribute for some event is missing or `UNKNOWN`, $\delta(e)$ for that attribute evaluates to FALSE, as an unknown value cannot reasonably match a known interval.

DEFINITION 1 (MATCH SCORE). *Given event e and subscription $\phi = \bigwedge_{i=1}^{n} \delta_i : w_i$, with $\delta_i = a_i \in [v_i, v_i']$, $score(\phi, e) = \sum_{i=1}^{n} w_i \mid \delta_i(e) =$ TRUE.*

For proration, the weight for an attribute is multiplied by the ratio of the size of the interval intersection to the size of the interval of the event. We note that proration is used as a confidence metric, and this practice can be extended to other confidence or similarity metrics.

DEFINITION 2 (PRORATED MATCH SCORE). *Given event $e = \{a_1 : [v_1, v_1'], \ldots, a_l : [v_l, v_l']\}$ and subscription $\phi = \bigwedge_{i=1}^{n} \delta_i : w_i$, with $\delta_i = a_i \in [v_i'', v_i''']$, $score(\phi, e) = \sum_{i=1}^{n} w_i \times \left(C + \frac{\textsc{Min}(v_i''', v_i') - \textsc{Max}(v_i, v_i'')}{v_i' - v_i} \right) \mid \delta_i(e) =$ TRUE. $C = 0$ for continuous intervals, and $C = 1$ for discrete intervals to account for the overlapping at the endpoints.*

The top-k matching problem consists in determining the top-k matching set, defined as follows:

DEFINITION 3 (TOP-k MATCHING SET). *Given a set of subscriptions Θ, and an event e, the weighted partial top-k matching set $\Phi \subseteq \Theta$ is defined as $\Phi = \{\phi \mid score(\phi, e) > 0 \wedge score(\phi, e) \geq score(\phi', e)\ \forall\ \phi' \in \Theta \setminus \Phi\}$ and $|\Phi| = k$.*

This definition does not specify handling of ties. For example, if Θ contains $k+1$ subscriptions yielding the same score for an event the implementation must pick k of those subscriptions. The definition also requires at least k subscriptions match each event with a positive score. Otherwise the implementation chooses between returning fewer than k results or including some results with scores of 0.

3.2 The Budget Window

In some systems which use top-k matching, advertisers specify a budget and a time period to serve their ads. The goal is to spend as close to the budget as possible in the given time when paying per match. If a subscription is matched too often, the budget is exceeded. If not matched enough, the advertiser is underserved. Bhalgat et al. [3] explain smooth delivery over the time window is preferred.

Hitting the target budget for a subscription is highly unlikely unless the scoring mechanism adjusts for the time window and budget. When a subscription is added, two values accompany it – the length of the window and the *budget*. The *begin time* is when the subscription is added, and amount *spent* is set to 0. The *end time* is *begin time*+*window length*. There is an additional optional configuration, $g(t)$, to specify the preferred matching distribution over time. $\int g(t)\,dt$ must be zero at *begin time* and monotonically increase to model ideal spending in the absence of negative spending. When not specified $g(t)$ defaults to $g(t) = 1$.

The multiplier must be less than 1 for subscriptions matching too often, and reduce their likelihood to be in the top-k. The opposite occurs for subscriptions not matching often enough. This leads to the following definition:

DEFINITION 4 (BUDGET WINDOW SCORE). *Given event e, subscription $\phi$and $score(\phi, e)$' from Definition 2,* $score(\phi, e) = score(\phi, e)' \times \frac{budget}{spent} \times \frac{\int_{begin\ time}^{current\ time} g(t)dt}{\int_{begin\ time}^{end\ time} g(t)dt}$

4. FX-TM ALGORITHM

This section presents our novel top-k matching algorithm.

4.1 Overview

Some existing approaches, including BE* trees [17], use a single data structure, and thus exhibit a worst case complexity linear in the number of subscriptions. The key strategy adopted by FX-TM is *partitioning by attribute*, i.e., to use one data structure *per attribute* (see Algorithms 1 and 2). We assume subscriptions from clients are stored on a *matcher*, which is a node to match events to subscriptions. Events arrive at the matcher which returns k matching subscriptions. A subscription is of the form $\phi = \delta_1 \wedge \cdots \wedge \delta_n$ where each δ_i is on a different attribute a_i. Every subscription ϕ is uniquely identifiable by *sid*. Weightings on elementary constraints of the attributes in the aggregation function can be either specified by subscriptions or specified by the event, with the latter option taking precedence. Proration can be used in either case.

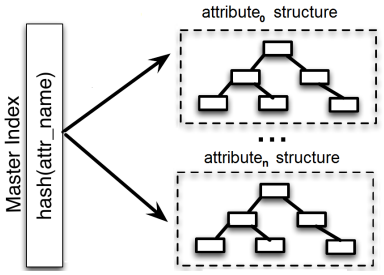


Figure 1: A representation of the two-level index.

In this section we refer to three main data structures which are used in FX-TM. The names of the relevant operations and their complexity bounds are in Table 1. Some interval tree operations may appear faster than the accepted operation, but they are achievable with some implementations, including that proposed by Arge and Vitter [2].

4.2 Subscription Partitioning

FX-TM employs a two-level index for subscriptions. Constraints on attributes are grouped together in an attribute-level index. A "master index" (Line 1) maps attribute names to (pointers to) data structures. Subscriptions are stored on matchers with predicates split by attribute name. Figure 1 shows a visual representation of the two-level approach. The structure for each attribute contains indexing information only for subscriptions containing that attribute.

We implement two types of structures for attributes. Attributes with discrete individual values use a hash map with the values as the keys and a tree set of matching subscriptions as the values. The tree set is ordered on subscription ids *sid* for quick insertion and deletion, but retrieval returns a list of all items in that tree. The second type of attribute structure is an interval tree for ranged attributes. Attributes are defined as ranged or exact matching so that the proper structure can be selected, and the selection must be consistent for all subscriptions with constraints on that attribute.

For example, a subscription `x = 'Indiana'` $\wedge$ `y`$\in$ `[60,100]` $\wedge$ `z` $\in$ `[0,2000]` is partitioned into its elementary constraints by attribute, and each of the constraints is stored in a separate attribute structure. Finding the structure into which a constraint on an attribute has to be inserted is done with the master index. Retrieving the top-k subscriptions that match an event is done by searching each of the relevant structures (possibly in parallel), and combining the results of the individual subscriptions. Thus, partitioning can be more advantageous when the matcher executes on multi-core machines or if each interval tree is on a separate node. Here we only consider single threaded single compute node matching because most state-of-the-art top-k matching algorithms do not consider concurrency or distribution; this ensures a fair empirical comparison of FX-TM against the same.

4.3 Adding and Removing Subscriptions

FX-TM employs one structure per attribute that exists in any subscription. FX-TM splits a subscription by attribute, and adds each constraint to the corresponding structure. If an existing subscription at a matcher has a constraint on a given attribute in a new subscription, then corresponding structure already exists in the master index (Line 8). The structure is retrieved (Line 7), and the constraint on the attribute is added (Line 9). If no existing subscription contains

Table 1: The data structures used in FX-TM with relevant function call names and execution time bounds.

Data Structure	Insertion	Deletion	Retrieval
Interval Trees[2]	TREE-INSERT: $O(log\, n)$	TREE-DELETE: $O(log\, n)$	GET-MATCHING-INTERVALS: $O(log\, n + s)$ for s matches
Tree Set[6]	TREESET-ADD: $O(log\, n)$	TREESET-REMOVE-MIN: $O(log\, n)$, TREESET-REMOVE-ID: $O(log\, n)$	TREESET-FIND-MIN: $O(log\, n)$, TREESET-GET-ALL: $O(n)$
Hash Map	HMAP-PUT: $O(1)$	HMAP-DELETE: $O(1)$	HMAP-GET: $O(1)$, GET-ALL-KEYS: $O(1)$ (plus traversal)

Algorithm 1 FX-TM algorithm: adding and removing subscriptions. For brevity, checking the attribute for ranged versus discrete values and using the appropriate structure is omitted in favor of assuming interval attributes.

```
1: masterIndex ← new hash map
2: budgetInfo ← new hash map
3: prorate ← TRUE/FALSE

4: upon RECV(ADD-SUBSCRIPTION, a_1 ∈ [v_1, v'_1] : w_1 ∧ ... ∧
   a_n ∈ [v_n, v'_n] : w_n, sid) do
5:    HMAP-PUT(budgetInfo, sid, φ)
6:    for i = 1..n do
7:       tree_i ← HMAP-GET(masterIndex, a_i)
8:       if tree_i ≠ ⊥ then
9:          TREE-INSERT(ROOT-OF(tree_i), [v_i, v'_i], w_i, sid)
10:      else
11:         tree_i ←new interval tree
12:         TREE-INSERT(ROOT-OF(tree_i), [v_i, v'_i], w_i, sid)
13:         HMAP-PUT(masterIndex, a_i, tree_i)

14: upon RECV(CANCEL-SUBSCRIPTION, a_1 ∈ [v_1, v'_1] : w_1 ∧
   ... ∧ a_n ∈ [v_n, v'_n] : w_n, sid) do
15:   HMAP-DELETE(budgetInfo, sid)
16:   for i = 1..n do
17:      tree_i ← HMAP-GET(masterIndex, a_i)
18:      TREE-DELETE(ROOT-OF(tree_i), [v_i, v'_i], w_i, sid)
19:      if SIZEOF(tree_i) = 0 then
20:         HMAP-DELETE(masterIndex, a_i)
```

a constraint on the attribute, a new structure is created for the attribute (Line 11 and inserted into the master index with a corresponding key (Line 13) after adding the weighted attribute-level constraint to it. Removing subscriptions is similarly straightforward. The structure corresponding to each attribute in the subscription is retrieved from the master index (Line 17). Each attribute-level constraint is deleted.

Empty structures may be removed from the master index. Partial matching only score attributes which exist in subscriptions *and* events. If an event contains an attribute with no existing structure, no subscriptions match on that attribute, so it does not affect matching.

A separate data structure indexed on *sid* for maintaining the information necessary to use the budget window multiplier is also updated (Lines 5 and 15).

4.4 Matching

We use summation as the aggregation function here, as that makes the most sense for weighted matching, but FX-TM supports all the aggregation functions of prior art.

FX-TM uses a hash map (*scoremap*) to track aggregate scores (Line 22), and a tree set to maintain a set of subscription identifiers sorted by their scores. FX-TM retrieves the set of subscription identifiers and weights for constraints matching each attribute. If proration is being used (Line 35), FX-TM computes the intersection of the attribute's value and the value associated with each constraint (Line 36).

The match score for each subscription is updated by querying *scoremap* with the subscription identifier and adding the (prorated) weight of the matched constraint (Lines 37 and 39). After matching all attributes for the event and updating the scores in *scoremap*, the entries in *scoremap* are adjusted by their budget window multipliers and pruned using the tree set *topscores*, which is ordered on match scores. Lines 40-49 show how entries from *scoremap* are inserted into *topscores* so that the size of *topscores* never exceeds k. Subscription identifiers and their associated match scores are added to *topscores* only if the subscription is among the top-k matches already seen in this phase, thereby ensuring that subscriptions known to not be in the top-k are never added to *topscores*. When the size of *topscores* would exceed k, the subscription with the lowest score is discarded.

When message producers specify weights on attributes, the weights from the subscriptions are discarded in Line 33. Otherwise the weights from the subscriptions are used.

5. COMPLEXITY ANALYSIS

Here we formally analyze the worst case time and space complexity of FX-TM. M is the number of attributes in an event or the number of constraints/attributes in a subscription. We assume M is the same for all N subscriptions and events. When that is not the case, the largest value for M determines a correct upper bound. We use interval trees for the analysis as in every case the $O()$ value is at least as large as the hash map required for discrete attributes.

THEOREM 1. *FX-TM handles* ADD-SUBSCRIPTION *operations in* $O(M \log N)$ *time.*

PROOF. From Algorithm 1. Insertion into an interval tree takes $O(log\, N)$ time (Lines 9,12). A subscription is inserted to M interval trees. Operations on $masterIndex$ (Lines 8, 7,13) take $O(1)$ time. Thus the total time complexity is $O(M \log N)$. □

THEOREM 2. *FX-TM handles* CANCEL-SUBSCRIPTION *operations in* $O(M \log N)$ *time.*

PROOF. From Algorithm 1. Deletion from an interval tree takes $O(log\, N)$ time (see Line 18). The subscription is removed from M interval trees. Operations on $masterIndex$ (Lines 17,20) take $O(1)$ time. Thus the total time complexity is $O(M \log N)$. □

THEOREM 3. *FX-TM performs matching in* $O(M \log N + S \log k)$ *time.*

PROOF. From Algorithm 2. GET-MATCHING-INTERVALS in Line 28 takes $O(log\, N + s_i)$ time, where s_i is the number of matching intervals for attribute a_i. Let $S = \sum_{i=1}^{M} s_i$, i.e., S is the total number of constraints matched across all attributes. To find matching intervals for each of the M

Algorithm 2 FX-TM algorithm: weighted partial matching of events to subscriptions.

21: **upon** RECV(MSG, $\{a_1 : [v_1, v'_1], \ldots, a_n : [v_n, v'_n]\}, \{a_1 : w_1, \ldots, a_n : w_n\}$) **do**
22: $scoremap \leftarrow$ new hash map {*Tracks match scores of partially matched subscriptions*}
23: $topscores \leftarrow$ new tree set {*Stores the results*}
24: $min \leftarrow 0$ {*Temporary variable used to store minimum score*}
25: **for** $i = 1..n$ **do** {*Loop through all attributes in event*}
26: $structure_i \leftarrow$ HMAP-GET($masterIndex, a_i$)
27: **if** TYPEOF(a_i) = $interval$ **then** {*Get matching interval attribute constraints*}
28: $matches \leftarrow$ GET-MATCHING-INTERVALS($structure_i, [v_i, v'_i]$)
29: **else** {*Get matching discrete attribute constraints*}
30: $matches \leftarrow$ TREESET-GET-ALL(HMAP-GET($structure_i, v_i$))
31: **for all** $\langle sid, [v_r, v'_r], w_r \rangle \in matches$ **do**
32: **if** $\exists w_j \neq 0, 1 \leq j \leq n$ **then**
33: $w_r \leftarrow w_i$ {*If weights are specified on the event, use those* }
34: $score \leftarrow$ HMAP-GET($scoremap, sid$)
35: **if** $prorate =$ TRUE **then**
36: $pwt \leftarrow$ PRORATE($w_r, [v_i, v'_i], [v_r, v'_r]$)
37: HMAP-PUT($scoremap, sid, score + pwt$)
38: **else**
39: HMAP-PUT($scoremap, sid, score + w_r$)
40: **for all** $sid \in$ GET-ALL-KEYS($scoremap$) **do**
41: $w \leftarrow$ HMAP-GET($scoremap, sid$)
42: **if** SIZE-OF($topscores$) $< k$ **then** {*The first k matches are the initial top-k*}
43: TREESET-ADD($topscores, sid, w \times$ BUDGETWINDOWMULTIPLIER(sid))
44: **if** $w \times$ BUDGETWINDOWMULTIPLIER(sid) $< min \vee min = 0$ **then**
45: $min \leftarrow w \times$ BUDGETWINDOWMULTIPLIER(sid)
46: **else if** $min < w \times$ BUDGETWINDOWMULTIPLIER(sid) **then** {*New score is > than at least one top-k previously seen*}
47: TREESET-REMOVE-MIN($topscores$) {*Remove min to maintain max size of k*}
48: TREESET-ADD($topscores, sid, w \times$ BUDGETWINDOWMULTIPLIER(sid))
49: $min \leftarrow$ TREESET-FIND-MIN($topscores$) {*Get min for future comparison*}
50: **for all** $sid \in$ GET-ALL-KEYS($topscores$) **do**
51: SEND(MATCHED, $\{a_1 : [v_1, v'_1], \ldots, a_n : [v_n, v'_n]\}, \{a_1 : w_1, \ldots, a_n : w_n\}$) to RECEIVER-OF($sid$)

52: **function** PRORATE($w, [v_{bl}, v_{br}], [v_{il}, v_{ir}]$)
53: **return** $\frac{\text{MIN}(v_{ir}, v_{br}) - \text{MAX}(v_{il}, v_{bl})}{v_{br} - v_{bl}}$ {*Fraction of event which overlaps. Add 1 in the case of inclusive integer intervals.*}

54: **function** BUDGETWINDOWMULTIPLIER(sid)
55: $\langle budget, spent, begin\ time, end\ time, g(t) \rangle \leftarrow$ HMAP-GET($budgetInfo, sid$)
56: **return** $\frac{budget}{spent} \times \frac{\int_{begin\ time}^{current\ time} g(t)\,dt}{\int_{begin\ time}^{end\ time} g(t)\,dt}$

attributes of an event, the total time complexity of querying the interval tree is thus $O(M \log N + S)$. Updating the subscription's score in *scoremap* is $O(1)$ (Lines 37 and 39) as is querying $masterIndex$ for interval trees (Line 26). $|scoremap| \leq S$. $|topscores| \leq k$, so inserting each subscription from *scoremap* to *topscores* takes $O(S \log k)$. There may be a maximum of $S - k$ removals and searches for the tree minimums, which has an additional time complexity bounded by $O(S \log k)$. There are S budget window updates with constant time each for a total of $O(S)$, which is less than $O(S \log k)$. Thus the total time complexity is $O(M \log N + S + S \log k)$, i.e., $O(M \log N + S \log k)$. □

THEOREM 4. *FX-TM requires $O(M\,N\, + k)$ space.*

PROOF. Each interval tree uses $O(N)$ space [2]. There are M interval trees. Each *scoremap* uses $O(N)$ space as no object can appear in it twice. The tree set *topscores* uses $O(k)$ space. Thus the total space complexity is $O(M\,N)$ for subscription storage and $O(N + k)$ for matching for a total of $O(M\,N + N + k)$, i.e., $O(M\,N + k)$. □

6. IMPLEMENTATION

We outline the implementation of FX-TM in a distributed middleware system.

6.1 Local Implementation

A local controller has two input streams – one for subscriptions and one for events. The controller parses requests (add subscription, remove subscription, get top-k matches) and the raw data contained within. The controller processes the request by updating the local data, including updating attribute structures and tracking matches for budget window calculations, and returning the matches if applicable. The top-k algorithm component has its own API for managing subscriptions and issuing top-k matching requests and is interchangeable. Subscriptions and events support lists of an arbitrary number of discrete and interval attributes.

6.2 Distributed Aggregation Overlay

There are two reasons distribution is desirable. Firstly, it increases parallel computation for lower system latency. Secondly, it allows scalability beyond the memory limitations of a single machine. We use the LOOM [7] generic aggregation subsystem for distributed application of FX-TM rather than a creating a specialized overlay or using Cao and Weng's work [4], which focuses on bandwidth rather than latency. We place the local top-k matching implementation on a set of nodes, then provide LOOM with a list of these nodes and a simple merge function which combines sets of top-k results from subsets of the data (see Figure 2). LOOM

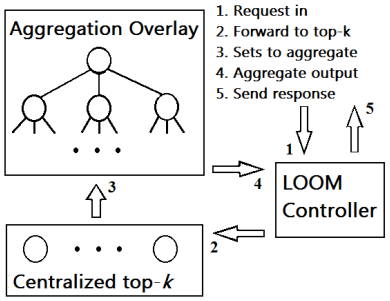


Figure 2: Overview of the full distributed system.

creates an aggregation hierarchy with a heuristically ideal fanout for minimal system latency based on the properties of the merging function. In this case of top-k the fanout is 3. The LOOM controller receives events for the system and forwards each event to every local controller to begin the matching process. The local controllers forward their results to the aggregation overlay, which returns the aggregate top-k results of the system. We use a simple script on the LOOM controller to distribute subscriptions evenly amongst nodes.

7. EVALUATION

This section empirically compares the performance of FX-TM to the current state-of-the-art. FX-TM is at least as expressive as state-of-the-art top-k matching algorithms, so we emphasize *matching time*, which is the time taken to retrieve the top-k subscriptions that match an event. Since our expectation is that the majority of a system's workload will be consumed by the matching of events rather than subscription additions and removals, our algorithm has relatively low theoretical bounds on addition and removal procedures, and the state-of-the-art alternatives are tuned for matching times, we focus on the matching time for the empirical evaluation.

7.1 Systems Compared and Setup

We test FX-TM against three alternatives. First we present an implementation of Fagin's algorithm [9, 10, 11]. Summation, our targeted aggregation method, is mentioned in the literature [9, 10], but in the presence of both positive and negative weights, it is not monotonic and thus not be supported. In our experiments, Fagin's algorithm uses $max()$, which is well covered in Fagin's literature. For an additional performance gain we use interval trees instead of a database backend for faster retrieval of stored constraints which overlap the event intervals. Matching intervals are retrieved from the trees and sorted in order of weight×prorated value, and Fagin's algorithm is applied to the resulting lists.

We also modify Fagin's algorithm to allow for summation with mixed weights without breaking monotonicity in an attempt to add greater expressiveness. The magnitude of most negative weight for each attribute is tracked. When an attribute is matched, all scores add that magnitude, including subscriptions which are not matched and have a natural score of 0. Thus no score is below 0, but the list for each contains all subscriptionsand must be sorted. The time to retrieve matches and sort the lists is presented as a lower bound on the execution time without actually matching as this time is already significantly longer than the other implementations.

Finally, we implement a BE* tree structure [17]. Rather than dynamically maintaining the structure as new subscriptions are added, we add all subscriptions to a temporary structure and then build the tree for all subscriptions. This creates a tree structure that has the desired attributes of the BE* tree for a static set of subscriptions. The resulting tree is ideal for lookup purposes. (Additions and removals after the initial setup, which we do not consider, require a complete rebuild of the tree.) In addition to the subtrees in a node for intervals which are left, right, and overlapping the partition value, we also have a subtree for subscriptions which do not include the partitioning attribute. When matching on intervals we prorate scores from subtrees with the largest possible intersecting interval in each subtree.

All algorithms are implemented in Java using a single thread. The distributed data access prior to aggregation assumed by Fagin [9] is easily translated into multi-threading that is also applicable to FX-TM with an appropriate locking scheme for concurrent updates and matches. [17] does not detail strategies for applying multi-threading, so single threaded sequential execution is the fairest comparison.

All centralized evaluations are executed on a machine with an Intel T6600 2.2Ghz processor and 4 GB of RAM. Each algorithm uses the same set of subscriptions and events for an experiment. Unless noted otherwise, averages and standard deviations are calculated from 1000 distinct matches.

Table 2: The default values for the experiments.

Variable	Generated Data	IMDB	Yahoo!
N	100000	100000	10000
k	1% of N 2% of N	1% of N 2% of N	1% of N 2% of N
M	12 per subscription or event out of 100 available	3 out of 3	5.4 (average) out of 22202
S/N	.22	.14	.11

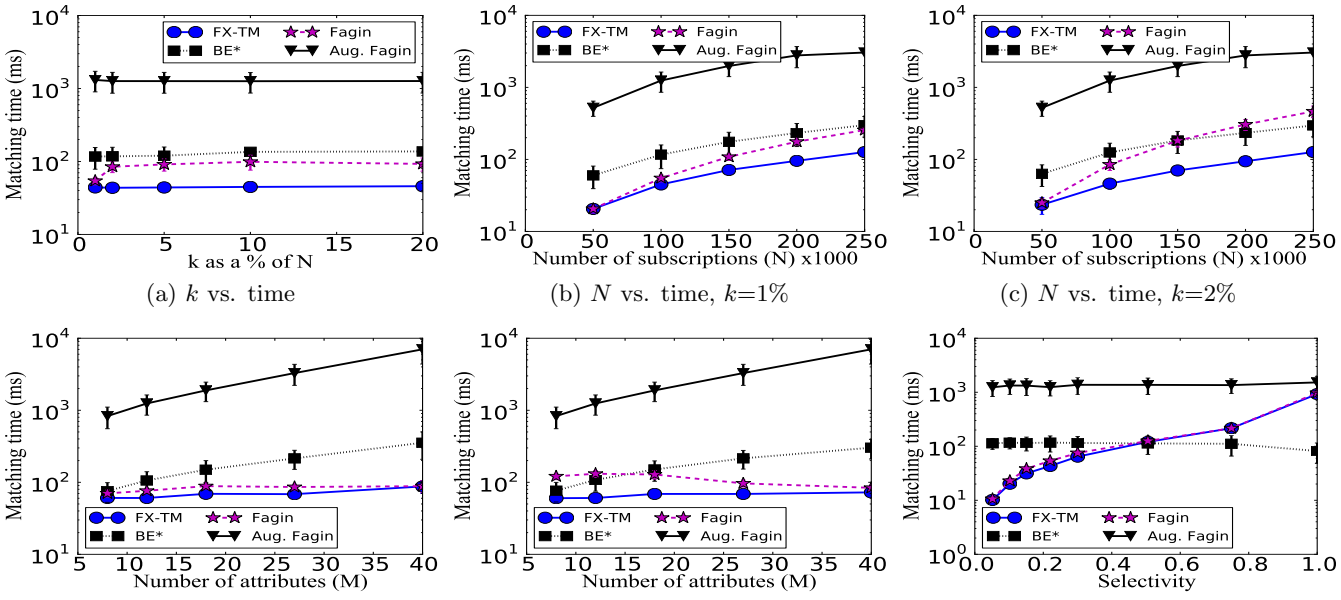
7.2 Micro-Benchmark Description

The variables which affect matching time of events are (1) the number of subscriptions (N), (2) k, (3) the number of attributes per event (M), and (4) the selectivity of the events for the subscriptions present. Selectivity is the fraction of subscriptions where the constraint on at least one attribute matches an event, i.e., S/N. We generate test data to record the effects of the variables independently. Default values, shown in Table 2, are reasonable but low enough that the effect of altering each variable is significant. S/N varies per event; the value shown is the average for the experiment.

The generated data contains positive and negative weights, as well as intervals which may overlap to either side for proration. This implements the full expressiveness achievable by FX-TM and a modified BE*. The standard Fagin algorithm uses a different aggregation method to allow it to run on the same data, so it returns a different set of top-k results. This is the only viable way to compare *performance* given the inherent inability to match the expressiveness.

7.3 Micro-Benchmark Results

The results for the micro-benchmarks are shown in Figure 3. Figure 3(a) shows how k affects the algorithms. FX-TM scales very well, which is expected given the $log\, k$ term in our theoretical bound. The BE* tree also scales well with k, which is indicative of a similar approach for holding intermediate results in a heap and not tailoring other parts of the



(a) k vs. time (b) N vs. time, k=1% (c) N vs. time, k=2%

(d) M vs. time, k=1% (e) M vs. time, k=2%. (f) Selectivity (S/N) vs. time

Figure 3: Results from varying each variable in the micro-benchmarks (logarithmic y-axis).

lookup process to k. Compared to FX-TM, the execution time is still increased by 165% to 200%.

As k impacts the amount of data sorted during Fagin's algorithm, k is expected to have a more dramatic affect on its runtime. Initially, that is the case as the execution time goes from 23% longer than FX-TM as k grows from 1% of N to 120% longer when k is 10% of N. The trend reverses slightly when k is increased further. This is because the number of matches per attribute hits a ceiling, limiting the length of the results to be sorted. The time for the augmented Fagin algorithm hits no such ceiling. As long as a single negative score exists in an attribute, the retrieval mechanism sorts all subscriptions for that attribute. The graph in Figure 3(a) shows this happens regardless of k, resulting in very slow executions times which are never within 2500% of FX-TM.

The effects of N on the algorithms are shown in Figures 3(b) and 3(c). Despite the logarithmic term in the theoretical bound for FX-TM the results look more linear. This is because to keep selectivity (S/N) fixed, S increases as well, which affects execution time. The performance of the BE* tree is slightly less sensitive to N and is 190% slower than FX-TM for small values of N and steadily improves to 137% longer for the largest value when k is 1% of N. When k is 2% of N, those numbers are 169% and 134% respectively.

Fagin's algorithm suffers worse as N increases. Higher values of N result in higher values of k, increasing the sorting time. The approach beats FX-TM by a fraction of a percent with low N and k= 1% of N. Increasing N to our maximum, though, results inversely in a 100% longer execution time than FX-TM. When k=2% of N, a 6% longer execution time than FX-TM grows to 167% longer when N is large.

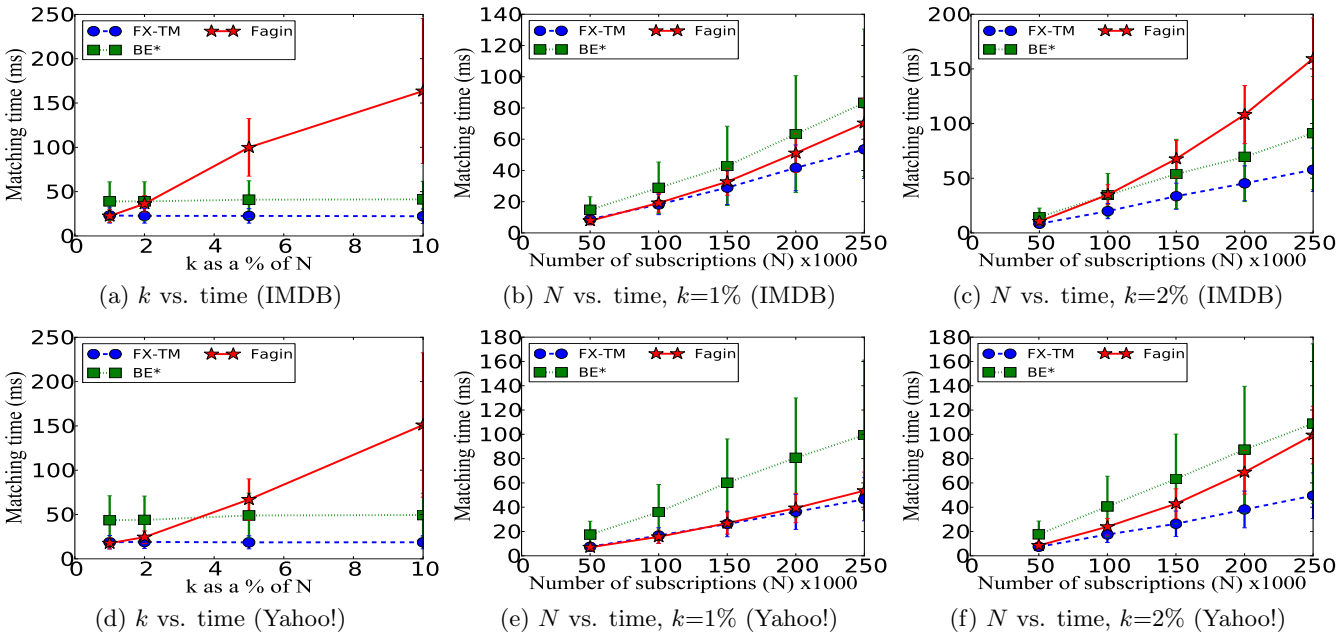
Figures 3(d) and 3(e) show how M affects the algorithms. There is almost no perceivable difference in the execution time of FX-TM, which indicates the low impact of M relative to the other variables. The effect on BE* is more noticeable as it takes 25% to 300% additional execution time compared to FX-TM for both values of k. This suggests that the effectiveness of pruning with single dimensions on a multidimensional index loses its potency when more dimensions vie with equal weight in a partial matching problem.

Figure 3(e) contains an anomaly for the execution time of Fagin's algorithm similar to that seen in Figure 3(a). Execution time decreases as M increases, bringing the execution time from a time accrued by 100% with respect to FX-TM for a low value of M to within 15% . This is counter to intuition and the trend seen in Figure 3(d). It is because the number of matches available for each attribute max out below k due to the nature of the data and the value of k. The problem is exacerbated as M grows, resulting in faster sorting times on smaller lists for each attribute. This happens for data with insufficiently high S/N as M or k grows.

The difference between Fagin's algorithm and an attempt to augment its expressiveness is clear. The performance of the original algorithm appears correlated to that of FX-TM, again indicating that data retrieval time is the bottleneck. The increase of execution time with M is virtually nonexistent. Yet, an increase in M, corresponding to an increase in the number of times each subscription is added to a list and sorted, results in a superlinear increase in execution time of the augmented Fagin algorithm.

Finally, Figure 3(f) shows how selectivity affects execution times. Decreasing S/N, which corresponds to decreasing S when N is fixed, shows an appreciable effect on the execution time of FX-TM. The BE* tree's multi-attribute index and clustering structure allows it to scale with selectivity better, but does not do as well for smaller values of S/N, which we believe are more representative for top-k. For very low S/N, BE* adds just over 1000% execution time to that of FX-TM, but it is 91% faster when S/N approaches 1. With our default values, the trade off point in performance between the algorithms is when S/N is about .5.

As with the other variables affecting mainly data retrieval times, the execution time of Fagin's algorithm is correlated to the execution of FX-TM. Fagin's algorithm does take more



(a) k vs. time (IMDB) (b) N vs. time, k=1% (IMDB) (c) N vs. time, k=2% (IMDB)

(d) k vs. time (Yahoo!) (e) N vs. time, k=1% (Yahoo!) (f) N vs. time, k=2% (Yahoo!)

Figure 4: Results from IMDB and Yahoo! real-world data benchmarks.

than 20% longer with values of S/N between .15 and .22, but at both extremes of the range for S/N the approaches are within 5% of each other. The time taken by the augmented variant of Fagin's algorithm is included in the graph for comparison, but there are no real trends. The existence of a single negative weight on an attribute requires every subscription to be included in that attribute's list prior to sorting, leading to an effective S/N value of 1.0 regardless of the actual selectivity.

7.4 Real-World Data Description

To confirm the relevance of our micro-benchmark results, we use two real-world datasets to generate subscriptions and events for more experiments. The first such dataset uses movie rating data from IMDB [13]. For each movie, IMDB provides the number of users who rated it and the average rating. We build small intervals around these values. The year of release is also provided. Thus all subscriptions and events have the same attributes. Subscriptions and events are generated the same way from different sections of the data. The best matches are subscriptions with similar voting patterns to an event and are released in the same year.

Our second real-world dataset is derived from the Yahoo! music rating service [20]. We use the same technique as in the IMDB dataset to build intervals around the number of voters and the average rating. Many songs also have anonymized genre and artist identifiers. These are discrete values. The best matches are subscriptions with similar voting patterns, matching genres, and the same artist as an event.

For each of these datasets, N and k are set to default values and varied as in the micro-benchmarks, but M and selectivity are features of the data. The values for each of the variables, including the defaults for N and k, are in Table 2.

7.5 Real-World Timing Results

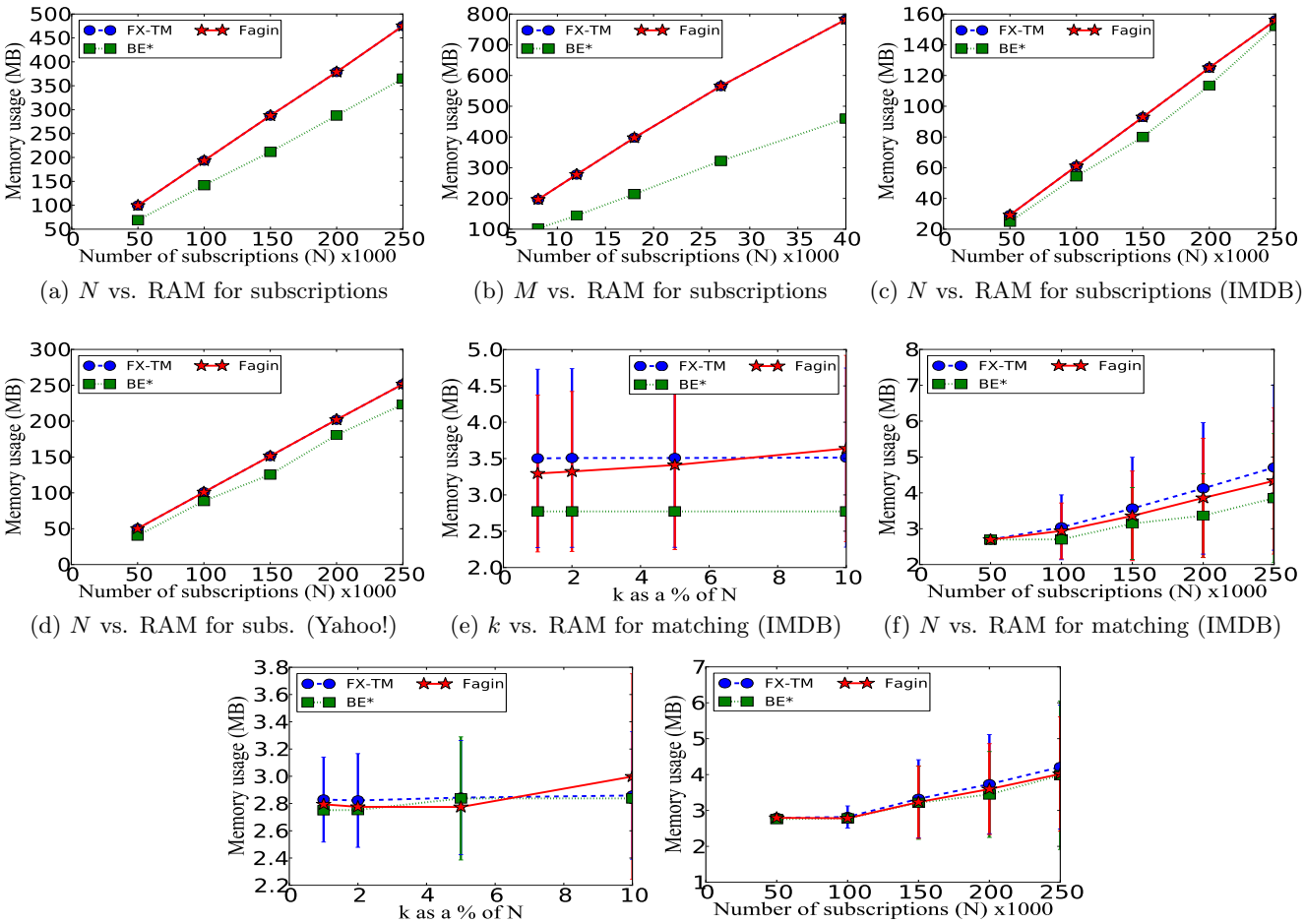
The trends using the real-world data, shown in Figure 4, mirror the trends in the micro-benchmarks. We omit the modified Fagin variant so the differences among the other algorithms is clearer. For the IMDB data, the BE* tree is never less than 48% slower than FX-TM in Figure 4(b). It is as much as 86% slower in Figure 4(a). This is a smaller difference than seen in the micro-benchmarks because M's value of 3 is smaller than the default value of 12 for the micro-benchmarks, and BE* is sensitive to M.

Fagin's algorithm is 10% faster than FX-TM when N and k are very low in Figure 4(b). When N or k is increased, that edge disappears. For our highest value of N, Fagin's algorithm takes 31% longer than FX-TM. Varying k in Figure 4(a) results in a quadratic change in execution time for Fagin's algorithm, indicating that the limit on the per-attribute list size seen in the micro-benchmark data is not seen here. When k is $N/10$ in Figure 4(a), the execution time for Fagin's algorithm is 636% longer than that of FX-TM. The general trends for N and k on all approaches are similar to those observed in the micro-benchmarks.

The Yahoo! dataset results also corroborate the micro-benchmarking results. Given that the value of M is in between the IMDB dataset and the lowest seen in the micro-benchmarks experiments, it is a little surprising that the execution time of the BE* tree is never less than 115% longer than FX-TM under the same conditions. This best case scenario is shown in Figure 4(e) when both N and k are relatively low. As those variables increase, the performance difference widens to a maximum of 164% in Figure 4(d). Fagin's algorithm is almost 7% faster than FX-TM for the lowest value of N tested in Figure 4(e), but adds 100% latency for our highest value of N in Figure 4(f) and more than 700% latency for the maximum value of k in Figure 4(d).

7.6 Memory Usage Results

In emphasizing the execution time, we expect to incur some overhead in memory requirements. We rerun two micro-benchmark tests and the real-world data tests to measure the memory used. We look at the memory used for storing



(a) N vs. RAM for subscriptions (b) M vs. RAM for subscriptions (c) N vs. RAM for subscriptions (IMDB)

(d) N vs. RAM for subs. (Yahoo!) (e) k vs. RAM for matching (IMDB) (f) N vs. RAM for matching (IMDB)

(g) k vs. RAM for matching (Yahoo!) (h) N vs. RAM for matching (Yahoo!)

Figure 5: Memory (RAM) usage used for storing subscriptions and for matching.

subscriptions and the space used by the matching algorithm per event. We first consider N, which affects the number of subscriptions stored and the tree indexing requirements. We also consider M, which affects the size of subscriptions and requires additional entries in attribute structures.

Memory use depends upon implementation of the data structures, so we realize that the exact values could vary widely. It is not advisable to draw conclusions about the direct comparisons of memory usage among algorithms, but the trends with relation to the variables and general orders of magnitude should be similar between implementations.

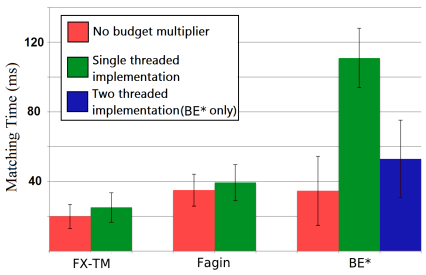
To find the amount of memory required to store subscriptions, we first initiate garbage collection and read the memory used by the Java runtime. We populate the structures for each algorithm and manually initiate garbage collection before taking another reading. The difference between the two readings is the memory used to store the subscriptions.

Figure 5(a) shows N versus the memory requirements to store subscriptions. Each subscription requires roughly the same amount of memory independent of the indexing used. The number of nodes in a tree is linear to the number of leaf nodes. Thus the amount of memory used by all approaches is linear on N. A BE* tree uses less space than multiple interval trees due to it's only using the most relevant indexes at each level instead of indexing every attribute of every subscription. The memory required for storing subscriptions is the same for FX-TM and Fagin's algorithm, as shown in Figures 5(a)–5(d), because we use equivalent structures.

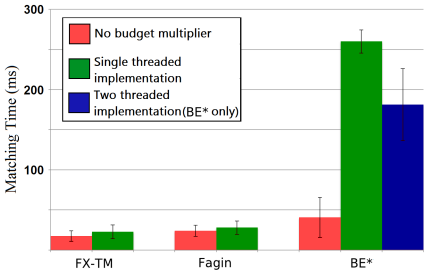
Figure 5(b) shows the effect of varying M on the amount of space required to store subscriptions. As each subscription stores its preferences, the size of subscriptions increases linearly with respect to M. Each new attribute also requires additional space in the indexing trees. In the case of interval trees which index each attribute, that can be expected to also have a linear relationship, which is supported by the data. In the case of BE* trees, the relationship is also linear.

Figures 5(c) and 5(d) show the impact of changing N on the storage space required for the data structures holding the real-world data. Both the IMDB and Yahoo! datasets show linear changes in storage space with respect to N, affirming the results from the micro-benchmark data. The Yahoo! dataset, with its higher value of M, requires more space than the IMDB dataset. This affirms the correlation of M and storage space seen in the micro-benchmarking data.

Matching requires additional memory. The amount depending on the event, we obtain an average reading for 500 different matches. We collect garbage between each match, and check our results to ensure that automatic garbage collection does not impact the numbers. The data collected is memory in use by the Java virtual machine beyond stor-



(a) IMDB dataset



(b) Yahoo! dataset

Figure 6: Overhead of the budget window mechanism.

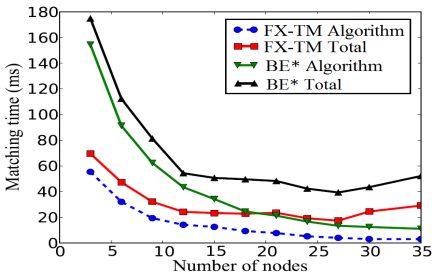


Figure 7: Results with LOOM.

ing the subscriptions, which includes memory used to match including function calls and temporary variables.

Intuitively, k, which determines the heap size in the approaches using heaps to maintain the top-k already found, should affect this memory usage. However, Figures 5(e) and 5(g) show this not to be the case. k has almost no discernible effect on the amount of space the matching took in either the Yahoo! or IMDB datasets. This is because the size of the heap itself is relatively inconsequential to the overhead incurred by the Java method calls and object creations.

The other variable expected to increase the space required for matching is S, which impacts the size of *scoremap* in FX-TM. For the real-world datasets with fixed S/N, that means increasing N. The results are in Figures 5(f) and 5(h). Increasing N from 50000 to 250000 results in an increase in memory usage of 75% FX-TM in the IMDB dataset and a 51% in the Yahoo! dataset. The other approaches also require more memory as N increases, but the increases are less significant. In none of these tests does the memory use exceed 5MB, which is at least an order of magnitude less than the space required to store subscriptions.

7.7 Budget Window Results

The next experiments test the impact of the budget window mechanism on matching time. To that end, each subscription is added a time window of $[1000000, 10000000]$ units and a budget of $[10000, 100000]$ matches. Every $g(t)$ is set to 1, making $\int g(t)\,dt = t - t_0$ for a homogeneous matching of the subscription across its time window. A time unit is the time taken by a single iteration of the matching algorithm.

To add the functionality to Fagin's algorithm, the multiplier is calculated in the same way as in FX-TM for each attribute before sorting. In both FX-TM and Fagin's algorithm, updates to the amount spent are made between matching iterations to maintain the single threading.

Implementing the mechanism in BE* is more cumbersome. The minimum and maximum multipliers must be propagated up the tree to inform pruning decisions. Since the multipliers change with time, they must be calculated and propagated continuously. We did this calculation and propagation in between each match. Since this process is clearly time intensive, we also moved the mechanism to a separate continuously running thread for a less onerous comparison to the other algorithms. In that case, pruning uses the current information at each level, which may be inconsistent with the state of the subscriptions further down.

For brevity, Figure 6 shows only the results of implementing the budget window mechanism in the real-world data sets using the default N value of 100000 and k at 2%. The first (red) bar in each group is the time taken by each algorithm without the budget window mechanism. The second (green) bar is the time taken when the mechanism is implemented within the same thread. For the BE* tree, the third (blue) bar is for the multithreaded approach.

The additional time taken for the budget window mechanism in the IMDB dataset is shown in Figure 6(a). FX-TM and Fagin's algorithm require an additional 25.6% and 12.5% respectively, with the overall time taken for FX-TM still being less than that taken by Fagin's algorithm. The time required with this feature incorporated into the BE* tree is significantly higher. Running completely on a single thread, the BE* tree leads to a 220% time increase. Running the update thread in parallel reduces this overhead to 53.1%, which is still significant compared to the other two approaches running in single threads.

The results with the Yahoo! dataset reiterate the previous results with more accentuated numbers. In this case, FX-TM requires a time increase of 29.7% over not implementing the budget window, and Fagin's algorithm requires 16.6% longer. The total time taken for FX-TM is still less overall than Fagin's algorithm. The BE* tree approach requires 540% longer when updating in the same thread and 346% longer when using a separate thread. This additional time indicates that the pruning mechanism during matching depends heavily on the data, and implementing the budget window mechanism can have a very large impact on the pruning.

7.8 Distributed Setup Results

With our JVM and machines, we are not able to run much more than 200000 subscriptions fitting the defaults from the generated microbenchmarks before running out of memory, which is insufficient for our target applications.

We evaluated the distributed setup on a group of blade servers at an IBM research center. We generated 500000 subscriptions using the defaults from the microbenchmarks. We evenly distributed the subscriptions across different numbers of leaves. Increased distribution decreases the load per leaf at the expense of greater aggregation. We tested each setup with 100 events and show the average times in Figure 7. Because we are trying to emulate a real system with full expressiveness, we only consider the two approaches which can handle non-monotonic score, i.e. BE* trees and FX-TM. For each algorithm we show two lines – the average amount of time taken for the local system at the leaves and the total amount of time from when an event enters the system to when the top-k results from the system are available.

Distributing the data decreases the effective N value at individual nodes, and the time for local computation decreases.

There is a point of diminishing returns as the relative addition of each new node is less significant, and aggregation time increases with the addition of more nodes. This is particularly noticeable as the number of nodes passes a threshold of a power of 3, which increases the height of the aggregation hierarchy. It is worth noting that while the aggregation time is generally the same between the two algorithms because the outputs from the local nodes are equivalent, it is slightly higher for BE* trees. This is because the variability of the local latency is higher, and the aggregation hierarchy has to receive all results to complete aggregation, so the maximum local computation time is as important as the average time.

For both algorithms the minimal total system time occurs when the data is distributed across 27 nodes. At this point the decrease in local computation time becomes less significant than the increased aggregation time for our data. At this point, the BE* tree takes 330% as long as FX-TM at the local nodes and 226% as long for the entire system. We note the total system time at 27 nodes is less than the local matching time when there were fewer than 12 nodes, so distribution effectively decreases total matching latency.

8. CONCLUSIONS AND FUTURE WORK

Expressive top-k matching with low latency represents a cornerstone for many "match-making" applications including online advertising, which has been driving a large economy based on online services. We introduce a new, more expressive, model for top-k matching which encompasses the models from prior art. It supports intervals in events and subscriptions with optional prorating for interval intersection. The model allows weights on events *or* subscriptions and supports non-monotonic aggregation.

We present FX-TM, an efficient and scalable algorithm for our model which runs in $O(M \log N + S \log k)$ time including data retrieval. This is experimentally competitive with the existing art, even without considering our extended expressiveness. Fagin's seminal algorithm is competitive for lower values of k as long as its limitations on expressiveness are acceptable. BE* trees, which can be modified to be as expressive as FX-TM, are shown by micro-benchmarks to be slower for selective data. For two real-world data sets BE* exhibited up to 2× higher matching time than FX-TM.

We present a dynamic mechanism to adjust the matching score based on the rate of matching and a predetermined budget and time window. With dynamic score adjustment, BE* requires up to 11× higher matching time than FX-TM.

We implement distributed top-k matching with LOOM for improved performance and greater scalability. This shows the relative impact of local algorithms on such a system.

We are considering ways to automatically detect the ideal degree of distribution and creating dynamic pricing models to adjust the price paid per match on the fly based on demand.

9. REFERENCES

[1] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching Events in a Content-based Subscription System. In *PODC*, 1999.

[2] L. Arge and J. Vitter. Optimal Dynamic Interval Management in External Memory. In *FOCS*, 1996.

[3] A. Bhalgat, J. Feldman, and V. Mirrokni. Online Allocation of Display Ads with Smooth Delivery. In *KDD*, 2012.

[4] P. Cao and Z. Wang. Efficient Top-K Query Calculation in Distributed Networks. In *PODC*, 2004.

[5] S.-C. Chu. Viral Advertising in Social Media: Participation in Facebook Groups and Responses among College-aged Users. *Journal of Interactive Advertising*, 12(1):30–43, 2011.

[6] T. Cormen, R. Rivest, C. Leiserson, and C. Stein. *Introduction to Algorithms*. MIT Press, 2011.

[7] W. Culhane, K. Kogan, C. Jayalath, and P. Eugster. LOOM: Optimal Aggregation Overlays for In-Memory Big Data Processing. In *HotCloud*, 2014.

[8] M. Dylla, I. Miliaraki, and M. Theobald. Top-k Query Processing in Probabilistic Databases with Non-materialized Views. In *ICDE*, 2013.

[9] R. Fagin. Combining Fuzzy Information from Multiple Systems. In *PODS*, 1996.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.

[11] R. Fagin and E. L. Wimmers. A Formula for Incorporating Weights into Scoring Rules. *Theoretical Computer Science*, 239(2):309 – 338, 2000.

[12] A. Goldfarb and C. E. Tucker. Online Advertising, Behavioral Targeting, and Privacy. *Commun. ACM*, 54(5):25–27, 2011.

[13] IMDB. IMDB Movie Ratings. `http://www.imdb.com/interfaces`.

[14] A. Machanavajjhala, E. Vee, M. Garofalakis, and J. Shanmugasundaram. Scalable Ranked Publish/Subscribe. In *VLDB*, 2008.

[15] M. McGowan. Facebook Rolling Out Video Ads to News Feeds Social Network Gives Brands Four Demographics to Target with 15-Second Spots. *Adweek.* `http://www.adweek.com/news/technology/facebook-rolling-out-video-ads-news-feeds-149239`, May 2013.

[16] M. Prigg. Dislike: Over HALF of Facebook Users Say they are Fed Up with Constant Adverts and Sponsored Posts. *Mail Online. (shortcut)* `http://unsourced.org/art/6978`, July 2012.

[17] M. Sadoghi and H. Jacobsen. Relevance Matters: Capitalizing on Less (Top-k Matching in Publish/Subscribe). In *ICDE*, 2012.

[18] C. Song, Z. Li, and T. Ge. Top-K Oracle: A New Way to Present Top-K Tuples for Uncertain Data. In *ICDE*, 2013.

[19] M. A. Stelzner. Social Media Marketing Industry Report. *Social Media Examiner.* `http://www.socialmediaexaminer.com/SocialMediaMarketingReport2011.pdf`, 2011.

[20] Yahoo! C15 - Yahoo! Music User Ratings of Musical Tracks, Albums, Artists and Genres v 1.0. Yahoo! Webscope `http://webscope.sandbox.yahoo.com`.

[21] J. Yan, N. Liu, G. Wang, W. Zhang, Y. Jiang, and Z. Chen. How Much Can Behavioral Targeting Help Online Advertising? In *WWW*, 2009.