Efficient Index Updates for Mixed Update and Query Loads

Sergey Nepomnyachiy
Computer Science & Eng.
New York University
sergey.n@nyu.edu

Torsten Suel
Computer Science & Eng.
New York University
torsten.suel@nyu.edu

***Abstract*—Inverted index files are commonly used to support keyword search in document collections. While the offline construction of an index can be done efficiently, its incremental update remains a hard problem, especially when the index does not completely fit into memory. We propose a novel approach for maintaining up-to-date index files on a system that constantly serves document updates and user queries. Unlike previous updating policies, we use knowledge of both the update term distribution and the query term distribution to partition the terms into functional groups. We implement two schemes for selective enforcement of contiguous layout of the data on disk, while mandating that the cost of the consolidation is less than its estimated benefit. The first is the "greedy merge" inspired by the ski-rental problem (as studied in the context of competitive analysis). The second is the "opportunistic prognosticator" – by making reliable predictions, the online problem becomes suitable for offline optimizations. We compare the performance of our system with base algorithms for incremental indexing in three different settings: an "append-only" system that grows indefinitely, and two fixed size systems where documents are deleted to accommodate new data, in either chronological order or in arbitrary order.**

I. Introduction

There is an ongoing and increasing need to perform full-text search on the vast amount of information generated today (e.g., web-pages, social media, logging) [1]. Users see a paramount importance in being able to search for fresh content as soon as it is introduced into the system. This mandates the transition of search systems from offline construction of their data structures (batch build) to online updates (incremental build) [2]–[4].

A ubiquitous data structure used to support full-text search is the *inverted index* [2], [5], [6]. An inverted index file contains *posting lists* for every term – ordered collections of references to all documents where the term appears [7], [8]. The ordering can be by document ids or by the expected importance (or impact) w.r.t. a given *ranking function* [9], [10]. In addition to the document id, it is common to store the number of appearances of the term in the document and their positions [11], which is used for advanced ranking and query techniques [12], [13].

When latency is important and the budget allows it, the inverted files are kept in the main memory [2], [5], [12], [14]. The large gap between main memory and hard drives (in price and performance) dictates that in many systems the inverted file is at least partially located on disk. In this paper, we focus on the latter case and do not consider incremental updates of an in-memory index as described in [2], [15].

General consensus for the architecture of an updating index introduces an intermediate sub-index in RAM – the *update buffer* (UB), sometimes called *delta index*. New documents are parsed and the produced postings are added to appropriate posting lists in the UB. At some point (e.g. when UB reaches a size limit) the data is flushed to disk in an *eviction phase* [11], [16]–[18].

Given a set of keywords, the query processing mechanism fetches the associated posting lists (usually in compressed form [11], [19]) and looks for documents where some (in OR queries) or all (in AND queries) of the terms occur. A ranking function is then used on the result set to order it by an impact factor. In this paper we make the conservative assumption that the posting lists are fetched entirely prior to processing, even for early terminating algorithms. While skipping parts of posting lists residing in main memory is beneficial, it is seldom the case for data on disk.

The job of the incremental indexer is to keep the posting lists fresh without degrading the efficiency of the query processor. A central challenge of disk-based storage is due to the limitations that the hardware imposes on the reader/writer of the data. Sequential access to data is fast, while random seeks take a lot of time. Therefore an inverted list placed contiguously on disk is read faster than a list occupying several segments. In Section II we show that in many of the proposed algorithms the core idea revolves around maintaining more or less contiguous lists, while incorporating new documents efficiently.

We claim that while it is important to keep some of the terms contiguous (e.g., the ones that are very frequent in the query log), for many terms the effort of consolidation is wasteful. We postulate that introducing a continuum of policies for scheduling merges, from "none at all" to "merge always", of small and uniform groups of terms can result in a better trade-off between index management and query processing time.

Contributions of This Paper: We propose a system for dynamic updates of inverted indexes that differs in how the evictions are made from a full update buffer in main memory, and in how and when the consolidation of segmented data occurs.

We partition the terms into functional groups using both update and query distribution estimates. To perform accurate optimizations, we impose a grouping of the terms into *'Term Packs'*. TPacks are non-overlapping sets of terms such that all members of a single TPack are comparable w.r.t. both update and query rate. The posting lists of a TPack are evicted together from the update buffer (UB), and hence are located in the same segments of the disk. Similarly, consolidation of segmented inverted lists is also TPack-wise: either all lists in a TPack are merged, or none.

We design and implement two strategies for scheduling merges of segments for the postings of the terms in TPacks. The *Opportunistic Prognosticator* uses the update and query distribution estimations to schedule optimizations when they are beneficial. The *Greedy Merge* resorts to a ski-rental based approach for scheduling optimizations when the future is unpredictable.

We demonstrate the merits of the system in three settings: an index with no deletion of documents, an index where old documents are deleted chronologically to accommodate new documents, and an index where random deletes are issued in roughly the same rate as the updates.

The remainder of this paper is organized as follows. In Section II we discuss the background and related work. The architecture of the proposed system is described in Section III. Section IV provides a detailed description of the experimental setup. In Section V we present the results of performance evaluation in multiple scenarios and discuss our results. Finally, Section VI concludes and discusses future work.

II. Background and Related Work

We outline and classify different methods for constructing and maintaining an up-to-date inverted index.

Offline Construction: During offline construction (or batch rebuild) of an index the raw documents are parsed to form postings of the inverted lists (document inversion). This process is illustrative of many processes that take place in an online inversion. Moreover, a periodic offline rebuilding of the index is the simplest form of handling incremental updates of the system (albeit the slowest of them). The techniques for offline construction and their efficient implementations are described in multiple surveys, including [5], [8], [11], [12], [20], [21]. In [21] Heinz and Zobel compare several popular variations of the inversion techniques, and propose an effective offline single-pass inversion method.

Re-Merge Updating: The naive approach to index updating is to rebuild it completely from the updated set of raw documents. This process is usually performed offline on a cluster of machines and the new index is then switched with the old one. An efficient index merging procedure (such as the one described in [21]) can be trivially applied to incremental updates. Under the re-merge strategy a batch of new documents is inverted in main memory and then merged with the large index file on disk. The immediate benefit over complete rebuild is avoiding unnecessary inversion of the old documents. However, the procedure is still costly, as we have to read and re-write the entire index file.

In-Place Updating: The in-place updating strategy is the opposite approach, where instead of handling the storage as a monolithic block, we operate on a term level. During an eviction from the update buffer (UB) the relevant terms are appended one at a time. When an inverted list exhausts the allocated space, it is transferred to a new area, usually with overallocation. For example, in [22] authors propose to reserve an extra 10% of space each time a list is relocated to accommodate future updates. This strategy requires multiple seeks to different locations on disk when postings of different terms are evicted from UB.

Comparison of Strategies: Lester, Zobel, Williams in [23] and later Zobel and Moffat in [11] compared the above approaches and concluded that if a single strategy is ultimately applied to all terms, *re-merge* is better. The intuition for this conclusion is as follows: to amortize the cost of multiple seeks of the in-place strategy, the system needs to buffer as many writes as possible before a disk-friendly write-back takes place. With small batches the seeks dominate the process, and with large batches most of the index is being read and written anyway.

Hybrid Strategies : In these systems the terms are divided into two classes – those that are handled with the re-merge strategy, and those that are maintained in-place [3], [17], [22], [24], [25]. The assignment of a strategy is usually governed by the length of the list, but also by its frequency in a query log [3], [25]. In [26], [27] the strategy applies to the entire index rather than to selected terms and depends on the nature of the workload in a time-window.

Non-Contiguous Layout: Both in-place and re-merge based systems go to extremes trying to keep the inverted lists contiguous. This is mainly due to the hardware limitations – hard drives are capable of reading sequential data extremely fast, while reading from random locations is very costly. Yet there are efficient systems where the contiguity demand is partially relaxed. In [25] a single monolithic inverted file is replaced with multiple sub-index files (horizontal partitioning). In [6], [28] a geometric partitioning is applied – the index is composed of several partitions with size growing geometrically. In [29] the authors implement two eviction strategies for the Proteus framework [30] – Selective Range Flush and Unified Range Flush. Under those polices the postings of long lists can occupy multiple segments. Similarly in [31], the long lists are realized as a blockwise linked list on a hard drive. In [27] the authors propose to apply a ski-rental [32] inspired technique to schedule the merging of the inverted file segments.

Summary: A system for incremental updates can be described by the following set of parameters:

- Inverted lists: strictly contiguous / segmented
- Space management: tight / over-allocating
- Terms partitioning: single set / multiple groups
- Consolidation policy: static policy / policy that changes dynamically at runtime

Under this nomenclature an always re-merging system from [23] is {strictly contiguous, tight, with a single set of terms, and static policy}; Lucene [6] is an example of {segmented, tight, single set, static policy}; Proteus from [29] is {segmented, over-allocating, multiple groups, static policy}; the system we propose is {segmented, tight, multiple groups (TPacks), dynamic policy}.

III. Proposed System

We propose a system where several standard components are redesigned to utilize the knowledge of update and query rates and distributions. In our system the update buffer (or delta index) is using a cost-based policy to make partial evictions and serves as a secondary term cache. An actual cache for posting list is also present and its size can be regulated dynamically, depending on the query rate. We allow for segmentation of the posting data and devise two dynamic policies for scheduling beneficial consolidations of the segments, again w.r.t. current rate and distribution of the update and query streams.

A. Motivation

Obtaining non-cached posting list data for a query incurs two costs – the penalties for random access ("seeks") and the cost of sequential transfer of the postings. The latter cost is unavoidable, but the seeks can be reduced to a minimum of one, if all the data is laid as a contiguous block. For every type of media (e.g., hard drives, SSDs) those costs are determined by the latency (in ms) and throughput (in MB/s) of the hardware. For each latency/throughput pair there exists a constant S s.t. for blocks larger than S the relative share of the seek penalty is insignificant.

To reduce the overall seek penalties the system is required to minimize the segmentation of the posting lists. The price of consolidations is bound from above by the most aggressive policy – always merging (AM). However, under high query rate most of the disk I/O is spent on fetching posting lists, while the consolidation cost becomes negligible.

Clearly, a policy fixed for all terms and query rate oblivious is always sub-optimal for some subset of the terms. This motivates us to introduce a system where the terms are divided into many functional groups according to their expected frequency in both the update and query stream. The consolidation scheduling for each group is not fixed, but governed by the cost effectiveness. This allows different parts of the index to progress through the continuum of policies dynamically.

To illustrate, if we learn that a group of terms participates in many queries we consolidate the posting lists of its members more aggressively, while terms that are rarely queried undergo almost no consolidation whatsoever. This contrary to eviction scheduling, where an often updated and queried group ought to be kept longer in main memory.

B. Architecture

We propose a system for dynamic updates of inverted index that is different in how the evictions are made from full UB and how and when the consolidation of the segmented data is scheduled. To be able to perform accurate optimizations we impose a grouping of the terms into "TPacks". TPacks are disjoint sets of terms such that all members of a single TPack are comparable w.r.t. both the update rate and the query rate. The postings of lists of a TPack are evicted together from UB, and are always located in the same segments of the disk. Similarly, consolidation of segmented inverted lists is also TPack-wise: either all lists in a TPack are merged, or none is.

1) Formal Definition of a TPack: We assume that for every term in our lexicon we have an estimation of their update and query frequencies (or probabilities). The update frequency of a term makes it possible to predict the length of its inverted list, while the query frequency allows us to estimate the number of queries that hit this term in a fixed time window. We define a TPack as a set of terms (members) such that:

- The terms have comparable update frequencies, i.e., the variance of lengths of the members is low.
- The variance of query frequencies predicted for the members is low as well.
- The sizes of posting lists of TPack members sum up to at least some threshold S.

The threshold S is chosen to be at least the amount of posting data we are able to read from disk during an average seek time. For such S the overhead of accessing the segment (i.e., paying for the seek) is never more than the time we need to read it entirely. Under a realistic distribution of terms in update and query streams and reasonable UB size and disk I/O parameters, the total number of TPacks varies from dozens to hundreds. The dynamics of TPacks follow a long-tailed distribution: we see a few TPacks with a small number of long lists and many TPacks with multiple short lists. In the extreme case we observe a TPack with a single frequent term versus a TPack with millions of singletons.

The notions of "frequently updated" and "frequently queried" are extended naturally from terms to TPacks. The "TPacking" of the terms creates sets with the uniformity that is inherently lacking when an entire term universe is considered. This uniformity allows us to apply fine-grained optimizations to the system.

We use a simple greedy assignment of terms to TPacks under the constraints stated in the rules. The terms are sorted by the update frequency first, with the query frequency applied for tie-breaking. The TPacks are formed by greedily assigning terms in the above-mentioned order until the expected threshold S is met. We find that partitions obtained with more rigorous techniques (e.g., clustering) do not improve the overall performance of the system.

The initial static assignment of terms to TPacks is executed using a distribution data obtained from a training corpus of updates and queries. We find that for documents coming from similar domains (e.g., training on TREC web pages [35] and running on ClueWeb web pages [36]) will mostly have similar distributions for updates and queries on a TPacks coarse granularity. With that said, with negligible overhead we prefer

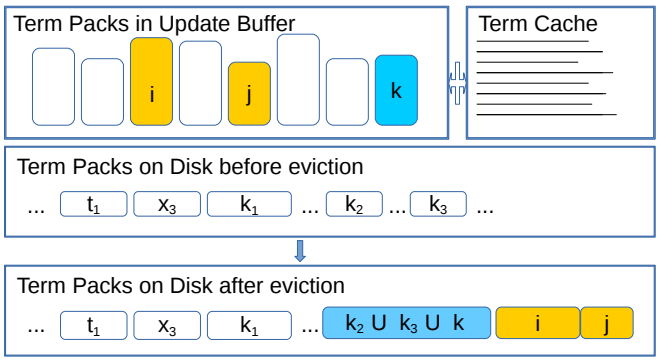

Fig. 1. System Architecture

to collect per-TPack statistics of updates and queries to allow for *re-packing* the terms if a skew in distributions is observed.

The overhead of TPacks is very small. We need to extend the lexicon to store a TPack id for every term, in our experiments a single byte was enough. Additionally for every TPack we store several counters accumulating update and query statistics as well as a vector of segment sizes. The length of those vectors is bound by the total number of evictions of UB.

2) Eviction: Unlike the monolithic re-merge policies, we do not evict the sub-index in the UB entirely. Instead, we only free a portion of the UB (e.g., 10%) by writing to disk the posting-data of TPacks that are less likely to be queried and updated in the future. This allows the terms (and TPacks) that are updated and queried frequently to benefit from remaining in main memory. Since the goal of an incremental index is to be able to always serve the most up-to-date information, the query processor is not limited to fetching the posting data from disk (or term cache), but also able to use the postings that were not yet evicted.

During an eviction of a TPack we write to disk one standalone segment containing the inverted lists of all members of this TPack in the UB. We demand that the evicted segment is at least of size S, however, it is expected (by design) that the TPacks reach this capacity during the filling of UB. We are different from systems that use fixed size segments [18], [29], [31], because we only constrain the minimal size of the evicted data. In fact, the rarely evicted TPacks tend to become very abundant in the UB.

In Figure 1 we demonstrate an eviction stage. In the partial eviction only three TPacks are chosen, depicted i, j, k. Here i and j are not considered for consolidation and written contiguously to disk. The posting data for k is consolidated with two of three existing segments on disk, forming a new segment – the union of k_2, k_3, k (but not k_1).

C. TPack Consolidation Policies

The price of merging a segmented index file is proportional to its length, while the benefit is proportional to the rate of queries that fetch it. Hence, it is possible that the "investment" in the consolidation of a term will not "pay off" in a finite time period, if it is not frequent in the query stream. In the experimental Section we show concrete examples of this for realistic distributions.

To consolidate a segmented file one needs to read all its postings, paying an additional seek for each segment, and then write the data back to disk contiguously. We claim that a system that applies static consolidation rules uniformly to all terms is bound to be sub-optimal. In our system the merges for each TPack are driven by a benefit function depended on the frequency of queries to it. All lists of the TPack are merged at the same time.

A consolidation of a TPack is considered beneficial if in the context of an experiment (or a chosen finite window, e.g., system uptime) the time spent on the merge is smaller than the sum of delays that are prevented in future queries.

1) Cost and Benefit: For M segments containing consecutive parts of the inverted lists with sizes $|s_1|, |s_2|, ..., |s_M|$ the consolidation price is:

$$CP = \sum |s_i| + M \cdot Seek + \sum |s_i| + Seek$$

i.e., traversing the data, paying for reads and M seeks, then writing it all to a new segment, paying for one write seek.

The benefit of that consolidation when Q is the number of queries that are expected to hit the segments in their lifetime is:

$$CB = (M-1) \cdot Q \cdot Seek$$

i.e., saving total of $M-1$ seeks for each of the Q queries.

When considering consolidation of two or more segments the following universal observations are driving our choices:

- Merging M segments into one eliminates $M-1$ seeks for all queries performed after the merge, regardless of the sizes of the segments, hence it is preferable to merge smaller segments to get better cost/benefit ratio.
- Delaying a beneficial merge reduces the benefit, since the cost of merge remains the same and the number of queries is diminished (when we consider a finite time frame).
- With that said, it is sometimes preferable to postpone a merge of existing segments until a new one is introduced to benefit from a N-way merge. For instance, instead of merging segments of sizes $2x$ and x we await for the arrival of another x-sized segment to form $4x$.

2) Greedy Ski-Rental: The scheduling of inverted file merges is an optimization problem that can be realized as a variant of the ski-rental problem [27], [32]. The ski-rental family of problems deals with scheduling optimization in systems where the future is unpredictable. In the non-randomized approach an optimization with a cost C is performed only after at least C worth of resources were "wasted" operating a non-optimized system.

In our implementation we keep an *overhead count* in the form of abstract tokens for each TPack – it relates to the number of superfluous seeks issued when serving queries to this TPack. A consolidation cost of a chain of segments is also expressed in tokens. A merge is scheduled for a TPack when it acquires enough tokens to pay for it. The cost of the merge at that point is deduced from TPack's tokens. This strategy

ensures a consolidation schedule well fitted for the needs of each TPack. Those with low-query rate are accumulating less tokens and rarely consolidate, while the frequently hit "afford" consolidating often.

Following the observations in III-C1 the system strives to spend as many tokens as possible on merging as many consecutive segments as possible. We start by considering the smallest segments (usually occurring in the tail of the segment list) and terminate the search when we get to unaffordable merges.

Note, that the accumulation of tokens is quadratic in the number of segments:

$$tokens = 1 \cdot q_1 + 2 \cdot q_2 + ... + m \cdot q_m$$

where q_i is the number of queries that hit the TPack between eviction $i+1$ and $i+2$ (when no merges were made). For non-decreasing query rate:

$$tokens = \sum_{i=1}^{m} i \cdot q_i \geq \sum_{i=1}^{m} i \cdot q_1 = O(m^2 \cdot q_1)$$

Yet, this does not mean that TPacks are easily driven to aggressive consolidation modes, since the cost for such policy also approaches quadratic. An explicit control of the aggressiveness is possible by introducing a conversion function between the I/O costs and tokens, thus making the algorithm more or less eager to perform consolidations.

Overall, this method aims at a decent trade-off between consolidation cost and query cost, reducing the number of "over-merged" and "under-merged" lists in the index.

3) Opportunistic Prognosticator: There exists a textbook efficient algorithm for optimal scheduling of merges [37], [38]. The list of segments is managed as minimum heap and the algorithm always fetches k smallest segments ($k \geq 2$) and puts their union as a new segment back to the heap. The process terminates when just one segment remains. This algorithm can be trivially extended to yield an optimal schedule given query counts. However, it can only guarantee optimality for the offline case, when all segment sizes and query counts are known ahead.

To utilize the power of the optimal algorithm we build a coarse TPack-level model of evictions to disk and query dynamics. The model uses the distributions obtained from training data and the statistics we gather for TPack during runtime. Using the model we can to some extent predict the near future and obtain an optimal scheduling for it via offline algorithm.

In listing 1 we show a pseudo-code of a procedure that predicts the sizes of segments for every TPack after E evictions of 10% of the UB, assuming no merges occurred. The only input it takes is the updates counter of each TPack.

In listing 2 we demonstrate how such prognosis is used to schedule consolidations. If a TPack has at least one segment on disk, when evicting another portion from UB we must choose whether to write it as a new segment or merge it with some (or all) of the existing segments. Note, that at this point the consolidation is cheaper, since the last segment still resides in main memory.

Algorithm 1 Prognosis

```
updateArray ← array of update counters
normalize updateArray
Segments ← []
for i = 1 to evictions − count do
    while UB not full do
        UB ≪ ∑ updateArray[i]
    end while
    while UB > 90% full do
        candidate ← leastPopularTPack()
        Segments[candidate] ≪ evict(candidate)
    end while
end for
```

We obtain a consolidation schedule optimal w.r.t. the prognosed segment sizes and the expected query counts. If in that schedule the new segment is destined to merge with one (or more) of its predecessors, we immediately perform the same merge, while the segment is still in memory. Otherwise, we write it as is. It is still possible that this segment participates in a merge with its successors in future.

Algorithm 2 Merge Scheduling

```
segments ← projected sizes of the segments
queries ← expected queries based on TPack stats
while |segmentSizesArray| > 1 do
    gain ← max (benefitAt(i) − costAt(i))
    if gain < 0 then return
    else
        i = argmax (benefitAt(i) − costAt(i))
        merged ← segments[i] + segments[i + 1]
        remove segments[i],segments[i + 1]
        insert merged at i
        record beneficial merge at i
    end if
end while
```

The crude model may be applied to alleviate another problem we face when designing a search engine – deciding how to use the valuable main memory budget. In particular, what portion of the memory ought to be devoted to the UB and how much will be used for the term cache (TC). General rule of thumb dictates to increase the size of the UB if updates are dominating the disk I/O of the system and favor the TC when most of disk I/O is due to queries hitting the system. Yet, finding the sweet spot for the UB/TC trade-off is tricky. Moreover, poor choice can significantly increase the disk I/O (as we are demonstrating in our experiments).

We propose to use the prognosis in order to pick the best combination of the two parameters: the UB to TC ratio and the portion of the UB we free during eviction. We start by assigning a reasonable value (e.g., $50/50$ for UB/TC ratio and 10% for eviction share, but after recording the update and

query statistics we use the model to perform parameter tuning.

4) Unreliable Predictions: We briefly list several factors that hinder our ability to perform precise predictions, and hence affect the performance of our algorithm.

- The query and update distribution of terms may change by either bursts or a slow drift, rendering our predictions unreliable. Although we are perfectly capable to recognize such discrepancy and refine our model it does not happen immediately.
- It is hard to account for the effects of a term cache on disk I/O. Ignoring it results in overestimating the needed disk I/O to popular TPacks and with it the benefit and the need for merges. To keep the model simple and fast we apply only very basic correction governed by expectations of cache hit ratio for each TPack.
- For the environments with document deletion we need to factor the churn rate when calculating the benefit of a merge.

In general, since the cost of collecting per-TPack stats of updates, queries, cache hits, etc., is insignificant, we prefer to constantly update our simple model and produce new predictions, rather than engineer a robust and expensive model.

5) Summary: We propose a system with term partitioning based on update and query frequencies and require that the UB is managed as a priority queue w.r.t. these partitions – evicting the least "interesting" data to disk first. We show two dynamic and query rate driven techniques for scheduling consolidation of segmented data.

IV. Experimental Evaluation

A. Studied Environments

We study the behavior of a dynamically updating index in three models of realistic usage. The index receives an interleaving stream of updates and queries. The queries are handled immediately, while the updates are applied to a small sub-index residing in main memory – update buffer (UB). The system uses an in-memory term cache (TC) for frequently queried terms to reduce the disk I/O for query processing.

1) Append only system – the updates are always additions of new documents. The inverted files in this case may grow indefinitely (in practice, until the disk space is exhausted).
2) Chronological deletes – to accommodate the data from new documents the postings of old documents are removed from the index in chronological order.
3) Arbitrary deletes – the documents are deleted in random order and at roughly the same rate as new ones are added.

In environments 2 and 3 we deal with steady-state systems – the disk space budget is limited, and the overflow is controlled by a churn-rate. In both cases we use the standard practice – maintaining a buffer of the removed content and applying the changes during merges [16], [39]. The setup can be naturally extended to support document revisions (such as in [40]).

B. System Prototype

We compare the performance of the proposed system with two approaches: full re-merge at every eviction (denoted AM) and logarithmic merge similar to the one used in Apache Lucene (denoted LM). The experimental prototype includes an implementation of the in-memory update buffer (UB) and a term cache (TC) using the Landlord algorithm for evictions [41], [42]. We assume that disk I/O is a dominant factor in any of the index operations and do not record the times of processing memory resident data (e.g., lexicon updates). Similarly, we omit the actual processing of the query for all algorithms and consider only the time it takes to bring the required inverted lists to memory (unless already cached).

The expected running time of many of our experiments is dozens or hundreds of days on a real hard drive, which renders them infeasible. Instead of actual disk I/O we direct all calls to simulated disk model that evaluates the required disk seeks and sequential R/W and translates those to time estimation. Similar simplifications were used in other works [22], [43].

We acknowledge that providing an accurate I/O model for multiple devices is challenging [44], [45] and limit ourself to overhead estimation with a simple model parametrized with average disk seek time (latency) and sequential R/W time (throughput). Our model is conservative and punishes non-sequential I/O (hence, our system).

A "lite" version of our system and simulator are publicly available at https://github.com/tpackindex/tpackindex (anonymized). It allows to perform crude but extremely fast estimations of running times of our system and two baselines under different query loads.

In our experiments a generic model of HD is configured with seek time of $7ms$ and 150 MB/s for sequential read. The configuration of the SSD model is based on Samsung 850 PRO drive characteristics [46] with latency of $0.06ms$ and sequential read of 500 MB/s.

C. Update and Query Streams

We stress our system with an interleaving stream of updates and queries. While it is relatively simple to produce a multi-billion posting stream of updates, the publicly available query logs are too small to provide similar load for the query stream. With that said, our goal is to measure the performance of the system in a realistic environment, hence it is crucial to mimic a stress of a real system. To achieve these goals we construct a language model of the updates and the queries.

We use TREC GOV2 dataset [35] (25.2 million web pages) and the first 80% of the TREC 2006 Efficiency track query trace as our training set, the remaining 20% are the test set. For each term we record the number of documents and queries where a term appears.

We also record the frequencies of terms in the documents of Clueweb09 TREC Category B [36] (50 million English pages). Next, we use MIT Language Modeling toolkit [47] to obtain from the training set, ClueWeb09 data, and the test set two distributions: update and query. Effectively, two non-zero probabilities are assigned to every term in the lexicon:

the chances to appear in an update and the chances to appear in a query.

The simulator samples these distributions to produce arbitrarily long update and query streams for our experiments. Note, that we avoid data pollution and do not make the "real" distributions explicitly available to the system. For the initial partitioning of the TPacks we use the training set, and the consolidation algorithms only use the distribution that they learn during bootstrapping phase.

We operate under the assumption that a query result cache exists in front of our system and only misses of that cache reach our engine. Similar implicit assumption is present in the query log we use, since it does not contain duplicate queries.

D. Setup

1) Envirnoments: We test separately the three supported environments (see Section IV-A). In the first phase of the experiment we stream $32GiB$ worth of updates to the system. Under all environments there are no deletions of postings during this phase, but the queries are processed in a normal fashion. The lengths of the inverted lists and the issued queries are the same for all algorithms at this point.

In the second phase we stream additional $224GiB$ of updates. For the environment with no deletes the index grows accordingly to accommodate all $256GiB$. In environments with chronological and random deletes the disk footprint of the index remains roughly $32GiB$ since we delete as much data as we add.

2) Query Rate: We stress the system with low, medium, and high query rates. The low query rate tests are to estimate the performance of the algorithms when the inverted file management is a significant part of the system running time, while the I/O for query processing is less influential. The high query rate is needed to show how different algorithms scale with the number of queries hitting the system, to uncover the different trade-offs inherent in them, and to estimate the running times in a situation where the I/O of the query processing component dominates the running time of the system.

Low: 16 queries per $1,000,000$ updates, total of $1,000,000$ queries. An always-merging algorithm with a UB of $2GiB$ spends 57% of disk I/O on queries.

Medium: 64 queries per $1,000,000$ updates, total of $4,000,000$ queries. An always-merging algorithm with a UB of $2GiB$ spends 84% of disk I/O on queries.

High: 1024 queries per $1,000,000$ updates, total of $64,000,000$ queries. An always-merging algorithm with a UB of $2GiB$ spends 98% of disk I/O on queries. The labels *low*, *medium*, *high* merely reflect on the relative share of disk I/O spent on processing queries and are not to be treated as absolute measures of query traffic in a real search engine. Note, that under all rates the term distribution of the query stream does not change – whether we issue a million queries or a billion, they all are sampled using the same language model.

3) Update Buffer vs. Term Cache: In all experiments $8GiB$ of main memory are shared between the update buffer (UB) and the term cache (TC). Larger UB results in less evictions and less segmentation. Larger cache reduces the disk I/O by serving popular queries from main memory. For every algorithm (apart from self regulating Prognosticator) we experiment using the following shares of the UB: $16\%, 25\%, 32\%, 50\%, 75\%, 96\%$. Since the performance of AM and our system degrades rapidly for UB share below 16%, we normally perform the tests with smaller UBs $(1\%, 2\%, 4\%, 8\%)$ on LM alone.

E. Comparing with URF

We implement the Unified Range Flush method from [29] as a fast representative of systems for incremental update. Similarly to our framework the terms are divided to groups and different policies are applied for managing them on disk. In URF a single group (range) contains consecutive terms ordered lexicographically. The number of terms in a range depends on the total space it occupies on disk. Upon overflow the range is split to multiple smaller ranges. The long lists are treated specially and managed using in-place strategy.

In [29] the authors present results for a very specific setup – a system with no deletions, no term cache, UB of 1GiB, and flushed memory size of 20MB. The authors use the standard corpora for documents and queries [35], [36]. While the build times are recorded for an incrementally updating index, the query time is measured for a single batch of 1000 queries after index construction. To be able to compare our system with URF we implement the algorithm for our simulator and execute it on an a same interleaving stream of queries and updates that was used for testing the base-line and our algorithms.

V. Results and Discussion

In figures 2 and 3 we present the results of experiments in Environment 1 (ever-growing) on HD and SSD respectively. AM shows the worst running time in both cases since there are not enough queries s.t. the benefits of contiguous lists outweigh the quadratic costs of the consolidation. In fact, this quadratic cost is visible in the chart for smaller UB shares. While increasing the UB share in memory significantly reduces the consolidation costs for AM, the total running time improves only upto 50% and slowly increases again as TC is reduced. This is simply because larger UB comes at the expense of the TC.

LM seems to be much more handicapped by decrease of TC share than benefiting from the growth of UB. This is only because it reaches its sweet-spot for UB shares much smaller than AM: $15\%-20\%$. For even smaller UBs the consolidations take their toll and total running time increases.

Our system with Greedy Merge powered by a ski-rental scheme (denoted SKI) performs better than either of the baselines. Like LM it sweet-spots on small UB shares. On HD it is as sensitive to TC deprivation in large UB cases, but for SSD its performance is much more stable. It is expected due to the discriminative eviction from UB – since the most popular TPacks are kept in memory and serve queries directly from

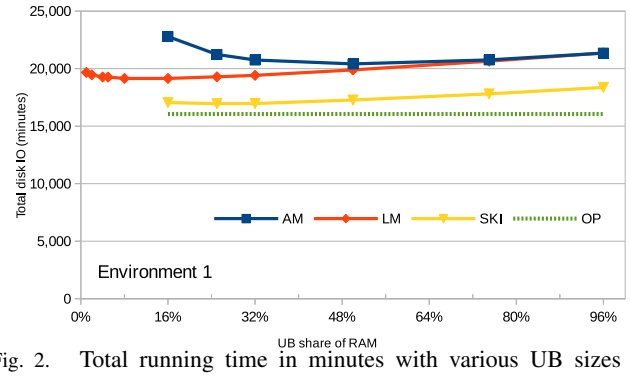

Fig. 2. Total running time in minutes with various UB sizes for medium query rate on HD. Lower is better.

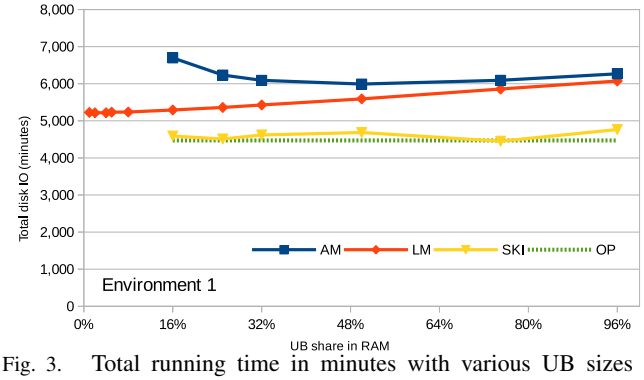

Fig. 3. Total running time in minutes with various UB sizes for medium query rate on SSD. Lower is better.

UB it effectively operates as a cost-based cache. In general, we observe that our system performs better on SSDs since the segmentation penalties are at least an order of magnitude smaller for it. LM also tends to benefit from it and shows larger relative gap in running time when compared to AM.

When we use the Opportunistic Prognosticator (denoted OP) the UB/TC ratio is chosen automatically, hence we present a single trend line to compare with other methods. We find that the calculated decisions OP makes result in better running time than the best of SKI on HD and as good as SKI on SSD.

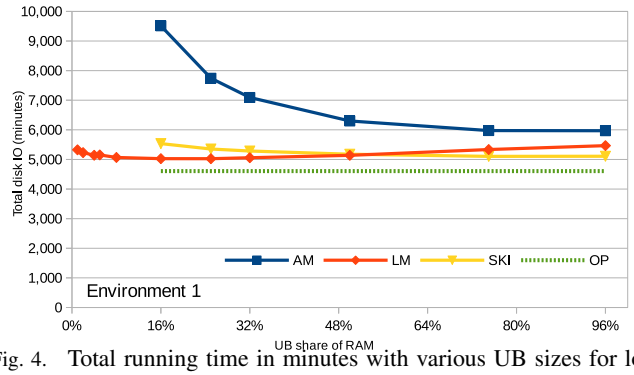

Fig. 4. Total running time in minutes with various UB sizes for low query rate on HD. Lower is better.

The results of experiments with low query rate are in Figure 4. The quadratic cost of consolidation is more profound for AM and it sweet-spots for UB share as large as 75%. SKI performs better than LM when UB is larger than TC and it is the other way around for smaller UBs. OP achieves better running times by allowing for a greater segmentation of TPacks and avoiding unhelpful merges.

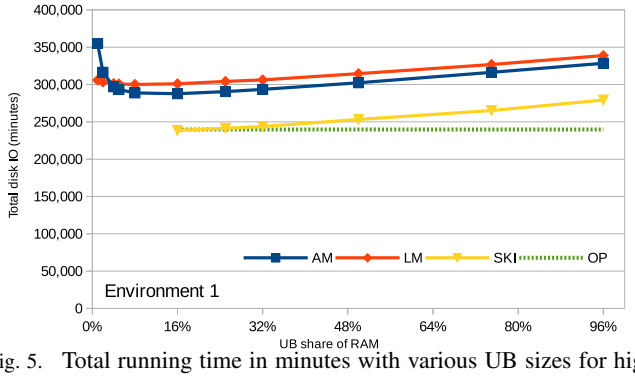

Fig. 5. Total running time in minutes with various UB sizes for high query rate on HD. Lower is better.

TABLE I
TOTAL DISK I/O TIME IN MINUTES FOR LOW AND HIGH QUERY RATES ON SSD. LOWER IS BETTER.

	Low Query Rate	High Query Rate
AM	1754	84,350
LM	1394	83,097
SKI	**1167**	**66,917**
OP	1260	68,007

In Figure 5 we observe the running time for high query rate. In this case the consolidation costs of all systems (even AM) become negligible and the importance of having a large TC becomes paramount. We see that in this case AM outperforms LM, since LM lists often require multiple seeks (logarithmic in evictions in the worst case). SKI performs much better than AM and LM (even its worst result is better than the best for AM). OP performs as well as the best result for SKI.

Due to lack of space we omit the charts for other experiments and present only the summaries with the best results for each algorithm. The charts in those cases show trends similar to shown in the above figures. The charts are publicly available at https://github.com/tpackindex/experiment-results (anonymous repository).

In Table I we show the best results for every algorithm for low and high query rate using SSD. On SSD we observe even better performance of both SKI and OP. We see that the model OP uses for future prognosis is too conservative for an SSD, causing a small bias in predictions. Consequently SKI shows slightly better performance in the best case. However, it is important to remember that obtaining an a priori UB/TC ratio to achieve the best result is not trivial.

In tables II and III we show the best results of experiments executed on Environment 2 and Environment 3 respectively. We find that our system outperforms the baseline algorithms in all experiments. Introducing deletions impedes our ability to make accurate predictions and in several cases we again observe better performance for SKI.

TABLE II
CHRONOLOGICAL DELETES. TOTAL DISK I/O TIME IN MINUTES. LOWER IS BETTER.

	HD			SSD		
	Low	Medium	High	Low	Medium	High
AM	1329	4605	65,562	365	1254	17,729
LM	1350	4982	77,353	333	1169	17,582
SKI	**1318**	4308	55,744	295	990	**13,910**
OP	1370	**4120**	**54,100**	**286**	**977**	13,922

TABLE III
RANDOM DELETES. TOTAL DISK I/O TIME IN MINUTES. LOWER IS BETTER.

	HD			SSD		
	Low	Medium	High	Low	Medium	High
AM	5075	18,732	283,770	1489	5494	83,191
LM	4892	19,040	301,714	1350	5146	83,061
SKI	5091	16,909	**238,478**	**1163**	**4447**	**66,745**
OP	**4587**	**16,033**	239,715	1256	4468	68,001

A. Comparing with URF

We run URF and our system with OP and the reference settings for URF: no deletion of documents, no term caching. Although in the original paper URF is handling only a single batch of 1000 queries after accommodating all postings, we stress it with our usual low, medium, and high query streams interleaving with updates. We run the experiment simulating a hard drive storage and UB budgets of 1 and 8 GiB.

TABLE IV
TOTAL DISK I/O TIMES (MINUTES) FOR URF AND OP

	UB: 1 GiB			UB: 8 GiB		
	low	medium	high	low	medium	high
URF	6700	23,000	353,500	5700	22,300	350,500
OP	6100	22,400	342,100	5000	18,600	287,300

A system based on Unified Range Flush shares similarities with our system. Data is only partially evicted from UB, the terms are grouped into functional sets, the segmentation of the index is controlled using a dynamic strategy.

However URF has no use for query distribution information. The partial eviction of data from UB does not necessarily favor less frequent lists. The grouping of terms is lexicographical and only long lists get special treatment. Moreover, the number of ranges constantly increases (reaching thousands in our tests), while the TPacks numbers are kept within manageable dozens. The segmentation control is dynamic, but it only accounts for the updates distribution.

We believe that by introducing query stream awareness to URF it possible to improve its performance to be on par or even better than the system we propose.

B. Summary

Our experimental results indicate that non-monolithic and query aware policies outperform the baseline approaches in most environments. Our system is less sensitive to bad choices of UB to TC ratios, especially on SSD. The Opportunistic Prognosticator often outperforms ski-rental based Greedy Merge or at least performs on par.

The results clearly imply that the query distribution and rate ought to be acknowledged when designing and configuring a search engine – whether for picking a proper eviction and consolidation policies, or for allocating main memory for UB and TC. We demonstrate that Greedy Merge is able to address the former and Opportunistic Prognosticator helps with the latter as well.

While showing that TPack-based system works in general, it is obvious that neither of our proposed schemes is always executing an optimal schedule. But we do believe that the results are promising enough to motivate further investigation of system reacting dynamically to update and query rates.

We observe that even though SSDs provide much better latency/throughput capabilities, the choice of an optimal policy is still very important, i.e., the index update problem remains actual. We see that the cheap seeks on SSD allow for more segmented data, which makes them extremely appealing to all systems that sacrifice contiguity (including logarithmic merge).

VI. CONCLUSIONS AND FUTURE WORK

We demonstrate that using competitive analysis and/or prediction for functional groups of terms often provides competitive edge over monolithic and static algorithms, while incurring insignificant overhead for book-keeping. The improvement comes from consolidation scheduling that strives to optimality, but also from turning the update buffer into another level of term cache with cost based eviction policy. Finally, we show that different choices of UB/TC ratios are beneficial in different scenarios and exploit this knowledge in Opportunistic Prognosticator.

In our future work we plan to further investigate the dynamic optimization of merge scheduling for TPacks. We plan to further improve the quality of prediction for the Prognosticator, thus allowing the system to make offline-algorithm quality decisions in an on-line setup. Finally, we want to test the system with real data, while writing the files to actual hard drives.

REFERENCES

[1] C. Sarigiannis, V. Plachouras, and R. Baeza-Yates, "A study of the impact of index updates on distributed query processing for web search," in *Advances in Information Retrieval*, 2009, pp. 595–602.

[2] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-time search at twitter," in *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 2012, pp. 1360–1369.

[3] R. Lempel, Y. Mass, S. Ofek-Koifman, D. Sheinwald, Y. Petruschka, and R. Sivan, "Just in time indexing for up to the second search," in *Proceedings of the 16th ACM International Conference on Information and Knowledge Management*. ACM, 2007, pp. 97–106.

[4] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications." in *OSDI*, vol. 10, 2010, pp. 1–15.

[5] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan, "Searching the web," *ACM Transactions on Internet Technology (TOIT)*, vol. 1, no. 1, pp. 2–43, 2001.

[6] A. Białecki, R. Muir, and G. Ingersoll, "Apache lucene 4," in *SIGIR 2012 Workshop on Open Source Information Retrieval*, 2012, pp. 17–24.

[7] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

[8] I. H. Witten, A. Moffat, and T. C. Bell, *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[9] E. Agichtein, E. Brill, and S. Dumais, "Improving web search ranking by incorporating user behavior information," in *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2006, pp. 19–26.

[10] W. B. Croft, D. Metzler, and T. Strohman, *Search engines: Information retrieval in practice*. Addison-Wesley, 2010.

[11] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM computing surveys (CSUR)*, vol. 38, no. 2, p. 6, 2006.

[12] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1, pp. 107–117, 1998.

[13] L. Wang, J. Lin, and D. Metzler, "A cascade ranking model for efficient ranked retrieval," in *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2011, pp. 105–114.

[14] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri, "Challenges on distributed web retrieval," in *Data Engineering (ICDE), 2007 IEEE 23rd International Conference on*. IEEE, 2007, pp. 6–20.

[15] N. Asadi, J. Lin, and M. Busch, "Dynamic memory allocation policies for postings in real-time twitter search," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2013, pp. 1186–1194.

[16] T. Chiueh and L. Huang, "Efficient real-time index updates in text retrieval systems," in *Experimental Computer Systems Lab, Department of Computer Science, State University of New*. Citeseer, 1999.

[17] D. Cutting and J. Pedersen, "Optimization for dynamic inverted index maintenance," in *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 1989, pp. 405–411.

[18] A. Tomasic, H. Garcia-Molina, and K. Shoens, *Incremental updates of inverted lists for text document retrieval*. ACM, 1994, vol. 23, no. 2.

[19] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering," in *Proceedings of the 18th International Conference on World Wide Web*. ACM, 2009, pp. 401–410.

[20] B. B. Cambazoglu and R. Baeza-Yates, "Scalability challenges in web search engines," *Synthesis Lectures on Information Concept, Retrieval, and Services*, vol. 7, no. 6, pp. 1–138, 2015.

[21] S. Heinz and J. Zobel, "Efficient single-pass index construction for text databases," *Journal of the American Society for Information Science and Technology*, vol. 54, no. 8, pp. 713–729, 2003.

[22] K. Shoens, A. Tomasic, and H. García-Molina, "Synthetic workload performance analysis of incremental updates," in *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Springer-Verlag New York, Inc., 1994, pp. 329–338.

[23] N. Lester, J. Zobel, and H. E. Williams, "In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems," in *Proceedings of the 27th Australasian Conference on Computer Science -Volume 26*. Australian Computer Society, Inc., 2004, pp. 15–23.

[24] S. Büttcher and C. L. Clarke, "Hybrid index maintenance for contiguous inverted lists," *Information Retrieval*, vol. 11, no. 3, pp. 175–207, 2008.

[25] S. Gurajada *et al.*, "On-line index maintenance using horizontal partitioning," in *Proceedings of the 18th ACM International Conference on Information and Knowledge Management*. ACM, 2009, pp. 435–444.

[26] S. Büttcher and C. L. Clarke, "Indexing time vs. query time: Tradeoffs in dynamic information retrieval systems," in *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*. ACM, 2005, pp. 317–318.

[27] J. P. MacCormick and F. D. McSherry, "Scheduling of index merges," Mar. 9 2010, uS Patent 7,676,513.

[28] N. Lester, A. Moffat, and J. Zobel, "Fast on-line index construction by geometric partitioning," in *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*. ACM, 2005, pp. 776–783.

[29] G. Margaritis and S. V. Anastasiadis, "Incremental text indexing for fast disk-based search," *ACM Transactions on the Web (TWEB)*, vol. 8, no. 3, p. 16, 2014.

[30] G. Margaritis and S. Anastasiadis, "Low-cost management of inverted files for online full-text search," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, 2009, pp. 455–464.

[31] E. W. Brown, J. P. Callan, and W. B. Croft, "Fast incremental indexing for full-text information retrieval," in *VLDB*, vol. 94, no. 66. Citeseer, 1994, p. 9.

[32] Z. Lotker, B. Patt-Shamir, and D. Rawitz, "Rent, lease, or buy: Randomized algorithms for multislope ski rental," *SIAM Journal on Discrete Mathematics*, vol. 26, no. 2, pp. 718–736, 2012.

[33] S. Krishnaprasad, "Uses and abuses of amdahl's law," *Journal of Computing Sciences in Colleges*, vol. 17, no. 2, pp. 288–293, 2001.

[34] G. K. Zipf, "Human behavior and the principle of least effort." 1949.

[35] "Trec gov2 page," http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm, accessed: 2014-11-11.

[36] "Clueweb09 dataset page," http://lemurproject.org/clueweb09/, accessed: 2014-11-11.

[37] D. E. Knuth, *The art of computer programming: sorting and searching*. Pearson Education, 1998, vol. 3.

[38] M. Schlumberger and J. Vuillemin, "Optimal disk merge patterns," *Acta Informatica*, vol. 3, no. 1, pp. 25–35, 1973. [Online]. Available: http://dx.doi.org/10.1007/BF00288649

[39] R. Guo, X. Cheng, H. Xu, and B. Wang, "Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree," in *Proceedings of the 16th ACM International Conference on Information and Knowledge Management*. ACM, 2007, pp. 751–760.

[40] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Agarwal, "Dynamic maintenance of web indexes using landmarks," in *Proceedings of the 12th International Conference on World Wide Web*, ser. WWW '03, 2003, pp. 102–111.

[41] Q. Gan and T. Suel, "Improved techniques for result caching in web search engines," in *Proceedings of the 18th international conference on World wide web*. ACM, 2009, pp. 431–440.

[42] N. E. Young, "On-line file caching," in *In Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM Press, 1998, pp. 82–86.

[43] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke, "Webbase: A repository of web pages," *Computer Networks*, vol. 33, no. 1, pp. 277–293, 2000.

[44] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101)," *Parallel Data Laboratory*, p. 26, 2008.

[45] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *Computer*, vol. 27, no. 3, pp. 17–28, 1994.

[46] Samsung. (2015) Ssd 850 pro. [Online]. Available: http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/ssd850pro/overview.html

[47] B.-J. Hsu and J. Glass, "Iterative language model estimation: efficient data structure & algorithms," in *Proceedings of Interspeech*, vol. 8, 2008, pp. 1–4.