Internet Mathematics Vol. 3, No. 2: 153-185

Inverted Index Support for Numeric Search

Marcus Fontoura, Ronny Lempel, Runping Qi, and Jason Zien

Abstract. Today's search engines are increasingly required to broaden their capabilities beyond free-text search. More complex features, such as supporting range constraints over numeric data, are becoming common; structured search over XML data will soon follow. This is particularly true in the enterprise search domain, where engines attempt to integrate data from the web and corporate knowledge portals with data residing in proprietary databases. In this paper we extend previous schemes by which an inverted-index-based search engine can efficiently support queries that contain numeric restrictions in addition to standard, free-text portions. Furthermore, we analyze both the known schemes and our extensions in terms of index-build time, index space, and query processing time. We show how to maximize query processing performance while respecting limits on index size and build time, or conversely, how to minimize index space and build time while maintaining guarantees on runtime performance. Thus, we concisely analyze the trade-off between index size and build time, and runtime performance. Finally, we present experimental results that demonstrate significant performance benefits attained by our method, as compared to alternative approaches.

1. Introduction

Both the search engine and database communities are increasingly focusing research efforts on integrating text search with structured and semi-structured data, in an attempt to narrow the gap between the two technological domains [Agrawal et al. 02, Bhalotia et al. 02, Gravano 01, Gravano et al. 03, Hristidis and Papakonstantinou 02, Raghavan and Garcia-Molina 01, Raghavan and Garcia-Molina 03]. Databases, which have traditionally managed structured data, are busy developing free-text search capabilities to efficiently and effectively handle queries over unstructured text columns in their records. Meanwhile, free-text search engines are broadening their scope to support semi-structured search over structured and semi-structured data. One particular example, which is the focus of this paper, is the increasing support in the search engine industry for searching over numeric fields, which we refer to as *numeric search*.

Recently, numeric search has become an important feature of free-text search engines. Product and shopping search applications such as Froogle and Yahoo! Shopping allow users, while entering a free-text search query describing a product, to also restrict the results to a certain price range. In eBay's advanced search, users can mix free-text search with numeric constraints such as the product's price range and the supplier's distance from a given zip code. In the advanced article search page of the *Los Angeles Times* website (as in most sites of major newspapers), users can restrict the set of articles that match a given free text query to those that were published within a certain range of dates. As the amount of semi-structured data increases—whether it is intrinsically supplied as part of the data or extracted from the data automatically—efficient algorithms for evaluating free-text queries with numeric constraints will become more important.

In this paper, we analyze and extend methods that augment inverted-index-based search engines with numeric search support. Note that these methods include the data representation aspect (indexing the numeric data associated with the documents) and the runtime aspect (accessing the text and numeric data when evaluating mixed queries). The main parameters of our analysis will mirror these aspects and will include (1) the time required to build the numeric portion of the index, (2) the space consumed by the numeric index, and (3) the time required to evaluate numeric constraints. Furthermore, we quantify the relationship between the first two quantities (index size and build time) and present a concise trade-off between the latter two quantities (index size and runtime performance).

Unlike known approaches in the literature to numeric search and related problems, our proposed scheme is completely insensitive to the type of numeric values found in the data (integer/floating point, positive/negative), to the distribution of values across the documents (clustered/nonclustered values), and to the granularity, or precision, requested by the queries. We thus make no assumptions on any of these issues.

The rest of this paper is organized as follows. Section 2 describes the types of queries we handle, introduces the inverted-index data structure, defines our model of a free-text query containing numeric constraints, and discusses the document-at-a-time (DAAT) evaluation model that is adopted in this paper.

Section 3 surveys related work. Section 4 discusses naive solutions to the numeric search problem, and Section 5 recounts the novel approach of Burrows to this problem [Burrows 96]. Sections 6 and 7 present our extensions to Burrows' work, analyze the performance trade-offs involved, and discuss implementation issues. Section 8 reports on experiments that we conducted with an implementation of our scheme inside an intranet search engine. We conclude in Section 9.

2. Preliminaries

2.1. Numeric Data and Queries

Our data model consists of a corpus of N text documents, where each document may also contain one or more numeric attributes, or *fields* (price, date,[1] age, longitude/latitude, etc.)

In general, documents may contain multiple values for some numeric field (e.g., a product in a catalog may come in several sizes). Alternatively, it could be that some documents do not contain any value for some numeric field. However, to ease the presentation, the analysis will mostly focus on the case where, for each numeric field, a single value is associated with each document. This allows our notations to be kept simple, with the symbol N denoting both the number of documents as well as the number of numeric $\langle$document,value$\rangle$ pairs to be indexed for each field. This simplifying assumption notwithstanding, the indexing and retrieval algorithms we present are easily extended to support multiple or no values per field per document, as explained in Section 6.4.

The queries that we address in this paper are those that contain both a free-text portion as well as constraints on the values of one or more numeric fields. Each such numeric constraint consists of an allowed range of values per some specific numeric field. Numeric constraints can be one-sided, defining ranges, $[v, \infty]$ or $[-\infty, v]$ (e.g., price $\leq$ 50), or two-sided, defining ranges, $[v_{min}, v_{max}]$ (e.g., 2001 $\leq$ year $\leq$ 2004). Two-sided queries are more general, and so most of the discussion in this paper will focus on those.

Note that queries in a search engine will seldom contain only numeric constraints; queries will typically contain a mixture of both free-text query terms and numeric constraints. For instance, a typical query could be "java book price $<$ 50," which has both textual and numeric constraints. We consider numeric constraints to be boolean restrictions, or *filters*, on the set of matching documents—we are not attempting to rank documents by their numeric values.

[1] One accepted numeric representation of dates is as the number of seconds since January 1st, 1970.

Rather, documents that respect the numeric constraints (and only them) will be ranked and returned according to the free-text portion of the query.

2.2. The Inverted Index

The *inverted index* (sometimes called inverted file) is the data structure of choice for full-text indexing in search engines [Arasu et al. 01, Witten et al. 99]. Algorithms for building inverted indexes are widely published [Baeza-Yates and Ribeiro-Neto 99, Brin and Page 98, Heinz and Zobel 03, Melnik et al. 01, Witten et al. 99]. Our goal is to develop a method that incorporates efficient numeric-search capabilities into the inverted-index model.

In an inverted index, each term or field appearing in the corpus is represented by a postings list. The set of all unique terms and numeric fields in the data is placed in a dictionary. Each dictionary entry points to a postings list, which contains a posting entry <docID, offset, payload> for each occurrence of the term in the corpus. The posting entries encode the *location* (docID and offset within document) of the occurrences and are typically sorted by this location. Indexing the offsets within the document enables the engine to support phrase queries and to perform proximity-based ranking of search results. In addition, an occurrence-specific *payload* is associated with each posting entry. Brin and Page used the payload to encode contextual information about the occurrence, such as the font size and capitalization [Brin and Page 98]. In general, for text tokens, the payload contains attributes used for ranking and display purposes. For numeric fields, the occurrence-specific payload may contain the value of the field.

Figure 1 shows our index structure. We assume that the index is stored on secondary storage and not in RAM and that each postings list is stored contiguously on disk.

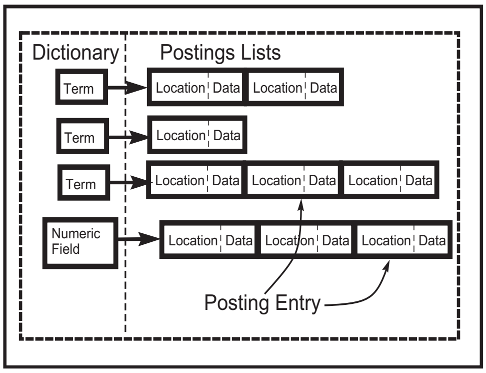

Figure 1. Index structure.

2.3. Document at a Time Evaluation

There are two main approaches to query evaluation using an inverted index. One is the *document-at-a-time model* (DAAT) [Brin and Page 98, Broder et al. 03], in which the postings lists of all query terms are traversed in parallel during query execution, identifying and scoring one candidate document after the other. The second is the *term-at-a-time model* (TAAT), in which the postings lists of the terms are traversed sequentially and a pool of accumulators is used to aggregate the contributions of the various terms to the candidate documents.

Each of the two models attempts to minimize the I/O required to evaluate queries. The TAAT model is useful when evaluating disjunctive (OR) queries, where the union of the documents containing any of the terms defines the set of matching documents. Since scoring the documents involves reading the entire postings lists of all the terms of the query, those lists are traversed sequentially, maximizing the amount of sequential I/O and minimizing the amount of disk seeks. On the other hand, the DAAT model is more efficient when evaluating conjunctive (AND) queries, where only the documents containing all query terms are considered as matching the query. This is particularly true in the presence of selective query terms (terms appearing in relatively few documents), as cursors on each postings list are moved efficiently to minimize disk I/O and optimize evaluation time. Here, the increased amount of disk seeks caused by the parallel traversal of different postings lists is offset by the much smaller amount of data that is actually read. Furthermore, DAAT evaluation naturally supports proximity-based scoring and is better suited for handling queries containing exact-match phrases.

The enormity of the web, coupled with the tendency of users to submit very short queries to web search engines, has caused conjunctive semantics to emerge as the de-facto standard in web and intranet search engines. Furthermore, as proximity plays a very important role in scoring short queries, DAAT is now the dominant evaluation model of web and intranet search engines. Note that DAAT is nicely suited to support our notion of numeric constraints as filters, presented in Section 2.1.

3. Related Work

Databases have been serving queries over numeric values for decades. Free-text queries have been served traditionally by full-text information retrieval systems and in the past decade by web search engines. In recent years there has been substantial research in augmenting databases with keyword search capabilities [Agrawal et al. 02, Bhalotia et al. 02, Gravano 01, Gravano et al. 03, Hristidis and Papakonstantinou 02] and in extending search engines to support richer query languages [Raghavan and Garcia-Molina 03].

Squeal is a system that allows SQL queries over web repositories [Spertus and Stein 00]. Its main goal is to support relational queries over web documents, while our approach allows for combined keyword and numeric search.

Recently, several XML processing algorithms that work over inverted indexes have been proposed [Bruno et al. 02, Carmel et al. 03, Kaushik et al. 04]. These algorithms use the information stored in postings lists to check structural constraints for XML queries. They also support the integration of XML and textual constraints and exploit these constraints in ranking [Carmel et al. 03]. However, none of these algorithms handle the parametric and numeric restrictions supported by XML query languages.

The problem of indexing numeric values using postings lists can be viewed as a bi-dimensional index problem, where the two dimensions are the location and the value. Several multi-dimensional index structures have been proposed in the database literature, such as UB-trees [Ramsak et al. 00] and R-trees [Guttman 84]. See [Samet 90] for a reference on the applications of spatial data structures.

To the best of our knowledge, the only previously disclosed work specifically addressing the support of numeric constraints in search engines is the patent filed by Burrows [Burrows 96] as part of his work on the core of the AltaVista search engine.[2] For deductive purposes, we defer the detailed description of Burrows' scheme to Section 5.

4. Naive Solutions

4.1. Single-List Approaches

Perhaps the easiest way to support numeric constraints in an inverted index is to create a single postings list for each numeric field. Like all postings lists, the entries in the numeric postings list will be ordered by docID, with each posting element storing the value that corresponds to the document. In order to simplify the presentation, we ignore the offset part of the location in the reminder of the paper.

During evaluation, the numeric postings list is filtered: query evaluation is driven by the free-text terms, and candidate documents matching the text portion of the query are filtered according to whether their numeric values match the given range constraints. We call this kind of numeric postings list a *filtered* postings list.

[2]AltaVista was then owned by Digital Equipment Corporation.

The solution above could be reasonably efficient when the free-text part of the query is selective (i.e., relatively few candidate results are passed through the numeric filter), but it is quite inefficient if the selectivity lies mostly in the numeric part of the query. In other words, the selectivity of the free-text part of the query serves as the lower bound to the number of numeric posting elements that must be scanned, regardless of the eventual size of the result set for the query. Since accessing each numeric posting may involve random I/O, filtered postings lists offer a simple indexing solution that comes at the cost of expensive runtime resources.

An alternative single-list approach holds the N ⟨docID, value⟩ pairs sorted by values rather than by docIDs. Here, two binary searches can identify the section of the list that holds values respecting the numeric restriction of a given query, thus identifying the documents that match the numeric part of the query. However, the matching documents are given in arbitrary order; they will need to be joined with the candidates determined by the free-text part of the query, which are typically produced in docID order. This is inefficient in cases where the numeric restriction is not highly selective. Furthermore, this join-based algorithm does not fit well within the architecture and flow of search engines.

4.2. Postings List per Distinct Value

A more advanced solution, which is efficient whenever the number of distinct values per field is small, is to have a dedicated postings list (naturally sorted by location) per value in each field. This solution also requires a sorted table of all distinct values per field, with each entry of the table pointing to the corresponding postings list (in other words, a dictionary of values for each field, sorted by values). During query evaluation, one consults the field's dictionary to determine the indexed values that fall within the requested range. Then, the corresponding postings lists are OR-ed (merged by docID) and AND-ed with the rest of the query's terms. Note that, in general, merging M postings lists involves reading in parallel from M different disk locations and becomes expensive as M grows. Therefore, to further optimize I/O in this approach, the various postings lists should be laid out contiguously on disk, sorted by the numeric value. Then, when sufficient RAM is available so that the lists can be merged in memory, the numeric part of each query requires reading a single contiguous area.

Despite the above optimization, this approach is not efficient for general value distributions in DAAT evaluation. In numeric fields containing a large number of unique values, the number of (often very short) postings lists that need to be OR-ed during evaluation grows. Furthermore, if the postings lists cannot be stored on disk contiguously by value, or cannot be read once and merged in-memory, performance may rapidly deteriorate as more distinct postings lists are accessed in parallel.

The scheme above is related to laying out the postings lists corresponding to text terms in lexicographic order, in order to efficiently support suffix completion, i.e., trailing wildcard queries such as "intern*" [Cutting and Pedersen 90].

5. Burrows' Scheme

Note that while the three simple approaches presented in Section 4 all suffer from inefficiencies in runtime performance, they index each $\langle$docID, value$\rangle$ pair once. Thus, the overhead in terms of index space is kept minimal. The breakthrough in Burrows' work [Burrows 96] involved improving runtime performance at the expense of additional preprocessing resources—indexing time and space.

Basically, Burrows proposes to re-index the same numeric data multiple times, naturally organizing the data differently each time. Specifically, Burrows suggests indexing numeric values in a *multilayered* scheme. He starts off by dedicating a postings list per distinct numeric value per field, as in Section 4.2: this set of postings can be thought of as being *layer*-0. He then builds additional layers of postings lists, with each layer being the result of merging pairs of adjacent-valued postings lists from the previous layer. Concretely, if layer 0 contained k postings lists for values $v_0 < v_1 < \ldots < v_{k-1}$, layer 1 will contain $k/2$ postings lists,[3] where layer-1 list j is the result of merging layer-0 lists $2j$ and $2j + 1$ by location (docID). This implies that the postings lists in each layer fully index the data, and hence the size of each layer is linear in the number of $\langle$docID, value$\rangle$ pairs, which we have denoted by N.

When one builds $\lceil \log_2 k \rceil - 1$ extra layers beyond layer 0, the last layer will contain two postings lists. Now, any one-sided query requires merging no more than a single list from each layer, and any two-sided query requires merging no more than two lists from each layer. Therefore, any numeric restriction requires the merging of $\Theta(\log(k))$ numeric postings lists, and some resiliency against multi-valued distributions is achieved. The price to be paid, of course, is the extra index space and index-build time required to build these extra layers.

6. Multilayered Schemes: Extension and Analysis

This section presents our main body of work, where we extend Burrows' multilayered scheme and concisely analyze the trade-off between index space and build time, and runtime performance.

[3]For simplicity, assume that k is even.

We begin by addressing several issues that arise from Burrows' scheme.

- Starting off with a postings list per distinct numeric value in layer 0 still leaves the scheme sensitive to the distribution of values. One may need to invest in many additional index layers until the number of postings lists to merge per query drops to acceptable levels. Furthermore, the size of the dictionary needed per field may be large—in the extreme case, the number of entries per field can equal the number of documents.

- Is the merging of pairs of adjacent postings lists from one layer to the next optimal? Could the merging of more than two postings result in better performance? What about changing the number of postings to merge within a layer? What about merging different numbers of postings as more layers are built?

- Denoting the number of extra index layers per field by L and the number of postings lists that may need to be merged per that field by M, can one generalize the trade-off between L and M, in a sense minimizing M given constraints on L and minimizing L given constraints on M?

Section 6.1 addresses the first issue, the structure of layer 0, and represents the first step toward our proposed solution. Sections 6.2 and 6.3 then explain and analyze our full scheme, and Section 6.4 discusses several implementation issues that arise from it. Section 7 presents a further extension, which is analyzed for the case of allowing a single extra layer of postings lists per numeric field.

6.1. A First Step: Equal-Sized Postings Lists of Increasing Value Ranges

As discussed in Section 4, when the number of distinct values present for some numeric field is large (when few documents share the same value), keeping a postings list for each value results in poor performance. The field's dictionary becomes quite large, and a query may require us to fetch many short postings lists and then to merge them by docID. In order to overcome such inefficiencies and to remove any dependency of runtime performance on the distribution of the values within the documents, we argue that *longer and fewer* postings lists should be created, by having each list ℓ cover a *range* of values, $r(\ell)$.

Specifically, we propose the following three-step process that creates b postings lists, each of size $\frac{N}{b}$, and two b-sized lookup tables, T_{min} and T_{max}. The two lookup tables will serve together as the dictionary of the field (thus limiting its size to $\Theta(b)$).

1. Sort all N $\langle$docID,value$\rangle$ pairs by value. Let V denote the set of distinct values.

2. Cut the single sorted list into b blocks of size $\frac{N}{b}$. For each block, populate T_{min} with the minimal value of the block (the value of the first item in the block), and T_{max} with the maximal value of the block (the value of the last item in the block).

3. Create a postings list from each of the b blocks, by re-sorting the elements within each block by docID. Note that while elements within a block are no longer sorted by value, for $1 \leq i < j \leq b$, all values in block i are no greater than any value in block j.

The complexity of the above procedure (index build time) is dominated by that of the first step, $\mathcal{O}(N \log N)$ if using a comparison-based sort or $\mathcal{O}(N)$ if using a fixed-width radix sort [Fontoura et al. 04]. The index space complexity is $\mathcal{O}(N)$, which is still minimal.

At runtime, upon receiving a query that requests a value range of $R = [v_{min}, v_{max}]$, evaluation proceeds according to the following three steps:

First step. Consult T_{min} and T_{max} (using two binary searches) to find the minimal and maximal list indexes, i_{min} and i_{max}, defined as follows:

$$i_{min} = \begin{cases} \max_{j \in \{0,\ldots,b-1\}}\{\, j : T_{max}[j] < v_{min}\} + 1 & \text{if } T_{max}[0] < v_{min}, \\ 0 & \text{if } T_{max}[0] \geq v_{min}; \end{cases} \tag{6.1}$$

$$i_{max} = \begin{cases} \min_{j \in \{0,\ldots,b-1\}}\{\, j : T_{min}[j] > v_{max}\} - 1 & \text{if } T_{min}[b-1] > v_{max}, \\ 0 & \text{if } T_{min}[b-1] \leq v_{max}. \end{cases} \tag{6.2}$$

It follows that whenever $i_{min} = b$, $i_{max} = -1$, or $i_{min} > i_{max}$, no documents match the query. In all other cases, the union of values in postings lists i_{min} through i_{max} includes all indexed values that fall within the range R. Formally,

$$R \cap V \subseteq \bigcup_{j=i_{min}}^{i_{max}} r(\ell_j).$$

Second step. Construct a *filtered* postings list from $\ell_{i_{min}}$ that only considers entries whose values are no smaller than v_{min}. Similarly, construct a filtered postings list from $\ell_{i_{max}}$ that only consider entries whose values are no greater than v_{max}. Denote these filtered lists by $f_{i_{min}}$ and $f_{i_{max}}$, respectively. Whenever $i_{min} = i_{max}$, the single postings list $\ell_{i_{min}}$ is filtered by both the lower bound v_{min} and the upper bound v_{max}.

Third step. Merge the following postings lists by docID, effectively OR-ing them:

$$f_{i_{min}}, \ell_{i_{min}+1}, \ldots, \ell_{i_{max}-1}, f_{i_{max}}.$$

Add the resulting postings list to the evaluation.

The above design, which is completely oblivious to the distribution and granularity of the numeric values, implies two worst-case guarantees on runtime performance:

1. The amount of filtering (length of filtered-postings) is bounded by $2\frac{N}{b}$ for two-sided queries.

2. The number of postings to merge is bounded by the number of blocks, b.

Clearly, we have a trade-off between the amount of filtering and the number of postings to be merged during query evaluation. Keeping both values low requires more resources, namely index space, as described in the next subsection.

The structure of the first step is similar in spirit to a B+Tree. The tables T_{min} and T_{max} fill the role of the internal nodes of the tree, while the b equal-sized postings lists can be seen as the leaves of the tree. However, there are notable differences: first, in B+Trees, the physical size of the leaves usually corresponds to the physical organization of data on a disk, whereas in our scheme the number of posting entries in each postings list ($\frac{N}{b}$) bounds the amount of filtering that is required (and does not correspond to the physical layout of the postings lists on disk). Second, the data in the leaves of a B+Tree and its internal nodes are sorted by the same key; in our scheme, the T-tables are sorted by numeric value, whereas the postings lists are sorted by locations (docIDs).

6.2. (L, c)-Canopy Design

In this section, we build a multilayered scheme on top of the b equal-sized (balanced) postings lists built in the previous subsection. The simplest way to achieve this would be to take the b postings as a balanced layer 0 and proceed to build additional layers as proposed by Burrows. We take this further by deriving the optimal manner to build the extra layers and by analyzing the trade-offs involved.

And so, with the b balanced postings comprising layer 0, let $c > 1$ and $L \geq 1$ be natural numbers: L denotes the number of *additional* layers of postings lists for the numeric field, and c will denote a *clustering* factor. For simplicity of exposition, we assume that c^L divides b.

The basic idea is that, for each layer j, $j = 1, \ldots, L$, we construct $\frac{b}{c^j}$ postings lists of size $\frac{Nc^j}{b}$. As in layer 0, each document appears in exactly one postings list of each layer. Postings list i, $i = 0, \ldots, \frac{b}{c^j} - 1$, of layer j is composed of the merging (by docID) of postings lists $ic, \ldots, (i+1)c - 1$ of layer $j - 1$. In other words, each layer clusters the postings lists of the previous layer in groups of size c, ending up with larger (but fewer) postings lists. We call this an (L, c)-*canopy* design, as the L layers of postings can be seen as the leaf layers of L B+Trees. However, these trees (as well as the tree of layer 0) all share the same "trunk": the two tables T_{min} and T_{max}. Therefore, rather than being a forest of $L + 1$ trees, the layers correspond to a large canopy of a single tree. Figure 2 shows an $(3, 2)$-canopy design, with three extra postings layers and a clustering factor 2.

6.2.1. Query evaluation at runtime. During runtime, the following steps are executed:

- Consult T_{min} and T_{max} as explained in Section 6.1. This implies which layer-0 postings lists define the range.
- Build filtered postings for the two extreme postings lists of layer 0.
- Select the appropriate postings lists from each layer so as to minimize the number of selected postings. We label the postings lists using a special numbering scheme in which each list in layer 0 is numbered sequentially and each higher-layer list is labeled using its layer-0-equivalent starting point (Figure 2). We choose the lists by a greedy iterative algorithm depicted in Figure 3. The intuition behind this algorithm is to use the longer postings lists from deeper layers as much as possible. A list from layer $i > 0$ will be selected if and only if its value range fully satisfies the query constraint, while some values of its containing list in level $i + 1$ fall beyond the query constraint.
- Merge the selected postings lists by docIDs, essentially OR-ing them via heap merge. The complexity of this step is $\mathcal{O}(\log m \sum_{i=1}^{m} |\ell_i|)$, where the m lists $\ell_1, \ldots, \ell_m$ are OR-ed and $|\ell_i|$ denotes the length of list i. Please note that this complexity is somewhat less than optimal if the lists have different sizes, in which case it could be better to recursively merge the two smallest lists until only one large list remains. Either of these two merge strategies can be applied in this step.

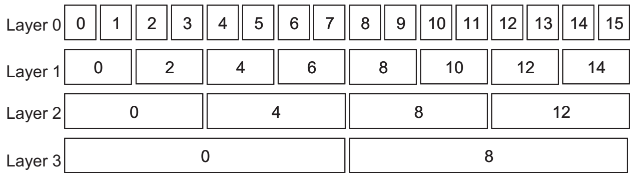

Figure 2. Numbering of postings lists for the list-selection algorithm.

```
void selectPostingsLists(
   int m,            // minimal layer-0 list index
   double v_min,   // the minimal value desired
   int M,           // maximal layer-0 list index
   double v_max,  // the maximal value desired
   int clustering) // clustering coefficient
{
   int from = m;
   int to = M;
   // We assume here that m < M. The case
   // where m equals M should be handled separately.
   // getPostingsList takes as input the list index
   // and layer and returns the appropriate list.
   b = getPostingsList(m,0);
   if ( v_min > getMinValueInList(b) ) {
      OrPostings.addFiltered(b);
      from=from+1;
   }
   b = getPostingsList(M,0);
   if ( v_max < getMaxValueInList(b) ) {
      OrPostings.addFiltered(b);
      to=to-1;
   }

   // Lists from...to should all be taken as is (no filtering).
   while (from ≤ to) {
      layer = dive(from, to, clustering);
      OrPostings.add(getPostingsList(from,layer));
      from += clustering^layer;
   }
}

// Helper function for list selection
int dive ( int fromListNum, int toListNum, int clustering ) {
   int layer = 0;
   int range = toListNum - fromListNum + 1;
   // Check if the layer+1 is a good candidate.
   while ( (clustering^(layer+1) divides fromListNum)
      && (range ≥ clustering^(layer+1))
      && ((layer+1) < L) ) {
      layer++;
   }
   return layer;
}
```

Figure 3. List-selection algorithm.

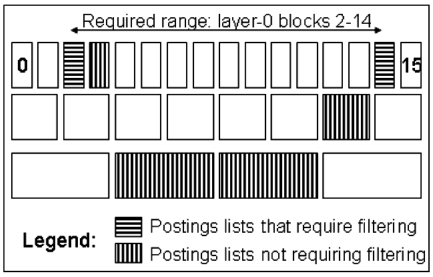
Required range: layer-0 blocks 2-14
0
15
Legend:
Postings lists that require filtering
Postings lists not requiring filtering

Figure 4. Postings list selection during query evaluation.

Figure 4 exemplifies the postings lists that are OR-ed for a query that ranges from layer-0 lists 2 to 14 in a $(2, 2)$-canopy scheme. Only six postings lists are accessed, covering the range of the 13 layer-0 lists. The role of the canopy should now be clear: the different sizes of the postings lists in each layer allow a numeric constraint to be satisfied with fewer, but longer, postings lists without increasing the amount of filtering required. The strict relationship between the sizes of the postings lists in the layers allows the canopy to share a single "trunk" (T_{min}, T_{max}) and allows the list-selection algorithm to efficiently select "leaves" from multiple layers to satisfy constraints.

6.2.2. Index build time. We now examine how the (L, c)-canopy postings lists are written to disk during indexing time. We discuss two schemes, depending on the amount of available RAM during the index build phase.

- **Inductive creation of layers.** Assume that we can fit $2N$ records in memory, giving us the ability to hold two layers simultaneously in memory. As explained in Section 6.1, creating layer 0 requires $\mathcal{O}(N \log N)$ time and is dominated by the initial sorting of the N $\langle$docID,value$\rangle$ pairs by value. Layer 0 can then be written contiguously to disk. The remaining L layers are created as follows: we hold two memory buffers of size N—one for the last layer already created and one for the layer being currently created. Given layer j, layer $j + 1$ can be prepared in time $\mathcal{O}(N \log c)$, as we heap-merge by docID c postings lists of layer j at a time. Layer $j + 1$ is then written to disk in a contiguous manner, one list at a time. Then, we build layer $j + 2$ using the buffer that previously held layer j. Overall, index build time is $\mathcal{O}(N[\log N + L \log c])$, where all layers are written sequentially without any intermediate disk seeks (and no reads at all).

- **Creation of layers from raw data.** If the available RAM can hold $3N + \Theta(b)$ records at once, we can create all $L + 1$ layers in $\mathcal{O}(N[\log N + L])$ as follows:

1. In the first N-sized buffer, we hold the N $\langle$docID,value$\rangle$ pairs sorted by docID.
2. We sort the N pairs by value into a second N-sized buffer and build the tables T_{min} and T_{max}.
3. Reusing the second buffer, we consult the T-tables and write, for every document d, the index of the layer-0 list that will hold the entry corresponding to d.
4. For each layer $j = 0, \ldots, L$, we logically divide the third N-sized buffer into the $\frac{b}{c^j}$ lists of layer j. All the lists are initially considered empty. We then scan the first buffer (the data, sorted by docID) and, for each $\langle d, v\rangle$ pair, fetch the layer-0 list corresponding to d, which we wrote in the second buffer. Let k denote that list number. The pair $\langle d, v\rangle$ belongs in list $\lfloor \frac{k}{c^j} \rfloor$ of layer j, and we write it to the first empty space in that list. Since we scan the data by docIDs, we are guaranteed that each layer-j list will be sorted by docIDs as well.
5. Each layer is written to disk in one motion.

Overall, sorting the N pairs by value and finding the layer-0 list corresponding to each document requires $\mathcal{O}(N[\log N + \log b]) = \mathcal{O}(N \log N)$. Then, each of the $L + 1$ layers is prepared in linear time, $\mathcal{O}(NL)$. Again, all layers are written sequentially without intermittent seeks.

We note that in RAM-limited environments it is also possible to build each layer efficiently (though with substantially more disk I/O) by using a sorted list (ordered by value) of size N on disk and making one pass over it to generate each layer using a RAM buffer of size $\frac{N}{b}$ that is accumulated, sorted by docID, and written to disk as needed.

6.3. Analyzing the Canopy Design

It is clear that a design with L layers introduces space demands of $\mathcal{O}(LN)$, as each layer includes entries for each of the N documents. While space demands grow linearly with L, the runtime's worst-case performance guarantees are reduced as follows:

- **Amount of Filtering.** The amount of filtering (length of filtered-postings) remains bounded by $2\frac{N}{b}$ for two-sided queries. Filtering is done at most on the two extreme postings lists of layer 0. Each such list is of size $\frac{N}{b}$.
- **Number of OR-ed postings lists.** The number of lists that should be OR-ed for this numeric term, denoted by $M(L, b, c)$, is bounded by $2L(c - 1) + \frac{b}{c^L}$: in

layer 0 we might need to OR $2c$ lists (c postings with low values, c postings with high values). From each layer $j = 1, \dots, L-1$, we need to consider no more than $2(c-1)$ lists. In layer L, we may need to OR all but two of the lists: $\frac{b}{c^L} - 2$. Summing the above,

$$\begin{aligned} M(L,b,c) &= 2c + 2(L-1)(c-1) + \frac{b}{c^L} - 2 \\ &= 2L(c-1) + \frac{b}{c^L}. \end{aligned}$$

In particular, when $L = \log_c b$, the number of postings lists to OR is reduced to $\mathcal{O}(c \log b)$.

The above formulation allows us to minimize $M(L,b,c)$ given constraints on the number of extra layers L and the amount of filtering tolerated, F. This is achieved by optimizing c.

Proposition 6.1. *The optimal value of c in an L-layered canopy, when filtering should be done on no more than F records, is $c_{opt}(L,b) = (\frac{b}{2})^{\frac{1}{L+1}}$ where $b = \frac{N}{2F}$.*

Proof. The filtering limit F implies that each postings list in layer 0 should contain no more than $\frac{F}{2}$ documents. This implies that the number of lists in layer 0 is $b = \frac{N}{2F}$.[4] Now, for given values of b and L,

$$\frac{\partial M(L,b,c)}{\partial c} = \frac{\partial[2L(c-1) + \frac{b}{c^L}]}{\partial c} = 2L - Lbc^{-(L+1)},$$

which is zero whenever $c = (\frac{b}{2})^{\frac{1}{L+1}}$. □

We can now substitute $c_{opt}(L,b)$ in $M(L,b,c)$, and we deduce that

$$\begin{aligned} M_{min}(L,b) &= 2L(\frac{b}{2})^{\frac{1}{L+1}} - 2L + 2(\frac{b}{2})^{\frac{L}{L+1}} \\ &= 2(\frac{b}{2})^{\frac{1}{L+1}}(L+1) - 2L. \end{aligned} \tag{6.3}$$

So far we have shown how to minimize M for tolerated values of b (or, equivalently, F) and L. Equation (6.3) readily allows us to maximize b (thereby minimizing F) given tolerated values of L and M.

Corollary 6.2. *The maximal value of b in an L-layered canopy where up to M OR-ed postings lists are tolerated is*

$$b_{max}(L,M) = 2\left[\frac{M+2L}{2L+2}\right]^{L+1}.$$

[4]For simplicity we assume that F divides N.

Finally, we note that minimizing L for tolerated values of M and b (calculating $L_{min}(M, b)$) can be achieved in $\Theta(\log L_{min})$ time using standard bracket and bisection schemes [Press et al. 88].

So far we have considered the clustering factor c to be the same across all layers. As noted at the beginning of Section 6, two questions arise:

1. Can allowing each layer to have its own clustering factor result in better performance? In particular, for given values of b and L, might L different clustering factors $c_1, \ldots, c_L$ yield better performance than just fixing a single value for c?

2. When building layer $k + 1$, can clustering the postings lists of layer k in groups of unequal size result in better performance?

The answer to the second question is yes, as will be shown in Section 7. However, as we subsequently show, the answer to the first question is negative: a uniform value of c is optimal.

In what follows, we assume that the product of $c_1, \ldots, c_L$ divides b. We denote by $M(L, b, \{c_1, \ldots, c_L\})$ the bound on the number of postings lists to be OR-ed when using an L-layered canopy with clustering coefficients $c_1, \ldots, c_L$.

Proposition 6.3. *For given values of L, b, and $c_1, \ldots, c_L$, let $\bar{c}$ denote the average of $c_1, \ldots, c_L$. Then,*

$$M(L, b, \bar{c}) \leq M(L, b, \{c_1, \ldots, c_L\}).$$

Proof. By considerations similar to those used above, we bound the number of lists that each of the L layers may contribute to the OR for this parameter. We deduce that

$$M(L, b, \{c_1, \ldots, c_L\}) = 2\sum_{j=1}^{L}(c_j - 1) + \frac{b}{\Pi_{j=1}^{L} c_j} \ .$$

Now,

$$\begin{aligned} & M(L, b, \{c_1, \ldots, c_L\}) - M(L, b, \bar{c}) \\ = \ & 2\sum_{j=1}^{L}(c_j - 1) + \frac{b}{\Pi_{j=1}^{L} c_j} - 2L(\bar{c} - 1) - \frac{b}{\bar{c}^L} \\ = \ & \frac{b}{\Pi_{j=1}^{L} c_j} - \frac{b}{\bar{c}^L} \geq 0. \end{aligned}$$

□

Another way to understand this result is to observe the symmetric roles of $c_1, \ldots, c_L$ in $M(L, b, \{c_1, \ldots, c_L\})$. Such symmetry intuitively suggests that all the c_is should be equal.

6.4. Final Implementation Notes

6.4.1. Omitting values in postings lists of layers other than layer 0. The postings of layer 0 are composed of ⟨docID, value⟩ ordered pairs. This allows for these postings lists to be filtered by value when some of the values in the list fall outside the numeric range R defined by the query. Postings of layers deeper than 0, however, are never filtered since they only participate in the evaluation when R fully contains the numeric range of the list. Therefore, the ⟨value⟩ component is redundant for the sake of retrieval and can be dropped if the application does not need the actual value, resulting in a more compact postings-list representation, in which each element carries no payload beyond the location.

6.4.2. Multiple values per field per document. In some cases, documents may contain multiple values for the same numeric field. For example, a page from a product catalog might describe products of different sizes. Our scheme is easily extended to support such cases, with minor changes both to implementation and analysis. Implementation-wise, in layer 0, distinct entries must be kept for multiple values of the same document: this enables filtering by value on the layer 0 postings lists. However, in all other layers, multiple occurrences of the same docID in a list are collapsed into a single entry. Actually, this is a direct consequence of the previous paragraph which proclaimed that only docIDs need be saved in deeper layers (the values are redundant there).

6.4.3. Value-based clustering of layer 0. So far, in order to bound the amount of filtering required by each query to $2F$, we have set the size of each layer 0 list to exactly $F = \frac{N}{b}$. In practice, however, it makes sense to impose another restriction on layer-0 lists: not allowing a single value to span across more than one postings list.

Figure 5 shows a simple algorithm that constructs layer 0 in a manner respecting both restrictions. Its input is the set of ⟨docID, value⟩ pairs, sorted by value.

Clearly, the algorithm does not allow a value to span multiple postings lists. Furthermore, any postings list containing two or more distinct values will surely be smaller than F, respecting the $2F$ bound on filtering. Note that the algorithm may create lists whose size is larger than F whenever a single value is shared by more than F documents. In such a case, a list will be dedicated to that value. Nevertheless, as large as they may become, these lists never need to undergo

```
// $B$ = the current postings list being constructed
// $V$ = the set of distinct values
// $B_v$ = set of documents associated with value $v$
$B \leftarrow \phi$
Foreach $v \in V$ (ordered by increasing values){
    if ($|B| + |B_v| \leq F$):
        $B \leftarrow B \cup B_v$
    else {
        write $B$ (sorted by docIDs)
        $B \leftarrow B_v$
    }
}
write $B$ (sorted by docIDs)
```

Figure 5. Value-based clustering algorithm of layer 0.

filtering. Their value either falls within the query's range, requiring the entire list to be taken, or it does not, rendering the whole list as irrelevant to the query. In addition, for equality constraints (where $v_{min} = v_{max} = v$), we never need to consider more than one postings list, thus eliminating the need of merging multiple lists.

This approach can potentially reduce the number of lists at layer 0, but it may also fragment layer 0 to include some small postings lists (with rare values sandwiched between two popular values). However, since every two adjacent postings lists are at least of size $F + 1$, the overall number of layer-0 lists is bounded by $2\lfloor \frac{N}{F+1} \rfloor + 1$.

7. Variable Clusterings within Layers

The canopy scheme discussed in the previous sections constructs an initial layer of postings lists, layer 0, whose size depends on the tolerable amount of filtering F. For each subsequent layer i ($i \geq 1$), it employs a clustering factor c_i that defines how many postings lists of layer $i-1$ are merged into each list of layer i. Proposition 6.3 proved that nothing can be gained by having different layers use different clustering factors; i.e., there is no reason to set $c_i \neq c_j$. However, we have yet to consider using different clustering factors *within* a single layer. This section shows that, in general, clustering variable amounts of layer $i-1$ postings lists when constructing level i results in better performance. Specifically, it typically lowers the bound on the number of postings lists that need to be OR-ed when evaluating a query.

We begin with an example, setting $L = 1$ and $b = 50$. The optimal canopy design with one extra layer has $M_{min}(1, 50) = 18$, achieved when setting $c = 5$,

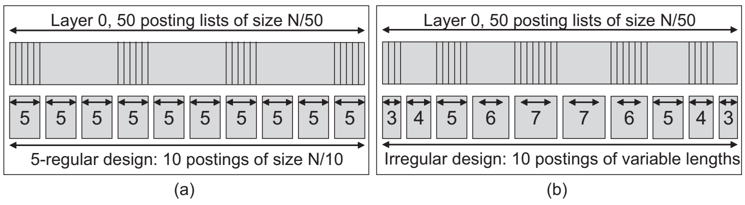
Layer 0, 50 posting lists of size N/50
5 5 5 5 5 5 5 5 5 5
5-regular design: 10 postings of size N/10
(a)
Layer 0, 50 posting lists of size N/50
3 4 5 6 7 7 6 5 4 3
Irregular design: 10 postings of variable lengths
(b)

Figure 6. Regular (a) versus variable (b) clustering factors.

as shown in Figure 6(a). However, clustering the 50 postings lists of layer 0 in groups of 3, 4, 5, 6, 7, 7, 6, 5, 4, and 3 lowers M to 14, as shown in Figure 6(b).

A full analysis of designs with variable clusterings within layers is left for future work. In this section, we analyze the case of a single extra layer ($L = 1$) for a given number of postings lists in layer 0, denoted by b.

Definition 7.1. A $\{c_1, c_2, \ldots, c_k\}$ *two-layered design* is a design in which layer 0 contains $b = \sum_{i=1}^{k} c_i$ blocks, layer 1 contains k blocks, and block i of layer 1 covers blocks $1 + \sum_{j=1}^{i-1} c_i$ through $\sum_{j=1}^{i} c_i$ of layer 0 ($c_i > 0$).

Definition 7.2. A $\{c_1, c_2, \ldots, c_k\}$ two-layered design is called M*-bounded* if any numeric query requires the merging of no more than M blocks.

It is easy to see that the following is true.

Lemma 7.3. *A $\{c_1, c_2, \ldots, c_k\}$ two-layered design is M-bounded if and only if the following two conditions hold:*

(a) For all $i = 1, \ldots, k$, $c_i \leq M$.

(b) For all i, j such that $1 \leq i < j \leq k$,

$$c_i + c_j + (j - i - 1) \leq M \ .$$

With the above definitions and characterization of M-bounded two-layered designs, we proceed to bounding b, given a design limit on M, the maximal number of postings to merge in any single query.

Definition 7.4. Let $b(M)$ denote the maximal value of b such that there exists an M-bounded two-layered design $\{c_1, c_2, \ldots, c_k\}$ with $\sum_{i=1}^{k} c_i = b$.

Proposition 7.5.

$$b(M) \geq \begin{cases} (\frac{M+1}{2})^2 & \textit{if } M \textit{ is odd,} \\ \frac{M}{2}(\frac{M}{2}+1) & \textit{if } M \textit{ is even.} \end{cases}$$

Proof. We begin by showing an M-bounded two-layered construction for odd values of M. Let $k = M$, and consider the design

$$\{1, 2, \ldots, \frac{M-1}{2}, \frac{M+1}{2}, \frac{M-1}{2}, \ldots, 2, 1\}.$$

First, by summing up these numbers, we deduce that layer 1 covers $(\frac{M+1}{2})^2$ layer-0 postings:

$$\frac{M+1}{2} + 2\sum_{i=1}^{\frac{M-1}{2}} i = \frac{M+1}{2} + \frac{M+1}{2}\frac{M-1}{2} = \left(\frac{M+1}{2}\right)^2.$$

Trivially, the first condition of Lemma 7.3 holds. For the second condition, we note that any two layer-1 postings that are separated by n postings cover no more than $M - n$ layer-0 blocks. In other words, for all i, j such that $j - i - 1 = n$, $c_i + c_j \leq M - n$. This implies that the second condition of Lemma 7.3 holds, and so the suggested design is indeed M-bounded and consequently $b(M) \geq (\frac{M+1}{2})^2$.

The construction for even values of M also has $k = M$, with the design being

$$\{1, 2, \ldots, \frac{M}{2}, \frac{M}{2}, \ldots, 2, 1\}.$$

Summing up the numbers gives $\frac{M}{2}(\frac{M}{2}+1)$, as claimed, and the two conditions of Lemma 7.3 hold in the same manner as for the case of odd values of M. □

Proposition 7.5 constructively lower-bounds $b(M)$. Before proceeding to show that the bound is tight, we present the following two lemmas.

Lemma 7.6. *Let $\{c_1, c_2, \ldots, c_{M-1}, c_M\}$ be an M-bounded two-layered design. Then,*

$$\sum_{i=1}^{M} c_i \leq \begin{cases} (\frac{M+1}{2})^2 & \textit{if } M \textit{ is odd,} \\ \\ \frac{M}{2}(\frac{M}{2}+1) & \textit{if } M \textit{ is even.} \end{cases}$$

Proof. We use the second part of Lemma 7.3 to deduce that $c_1 = c_M = 1$, otherwise $c_1 + c_M + (M-2) > M$ and the design is not M-bounded. Now, with

$c_1 = c_M = 1$, the same lemma also implies that, for all $j = 2, \ldots, M-1$,

$$\begin{aligned} c_j &\le \min\{M-1-(j-2), M-1-(M-j-1)\} \\ &= \min\{M-j+1, j\}. \end{aligned} \tag{7.1}$$

In fact, Equation (7.1) holds also for $j = 1, M$. Therefore, for even values of M,

$$\begin{aligned} \sum_{j=1}^{M} c_j &= \sum_{j=1}^{M/2} c_j + \sum_{j=1+M/2}^{M} c_j \\ &\le 2\sum_{j=1}^{M/2} j = \frac{M}{2}\left(\frac{M}{2}+1\right), \end{aligned}$$

and the proof for odd values of M is similar.

Note that the construction in Proposition 7.5 indeed defined $c_j = \min\{M - j+1, j\}$ for all $j = 1, \ldots, M$. □

Lemma 7.7. *Let $\{c_1, \ldots, c_k\}$ be an M-bounded two-layered design such that $2 \le k < M$. Then, either $c_1 = c_k = 1$ or one can construct a two-layered M-bounded design $\{d_0, \ldots, d_k\}$ such that*

$$\sum_{i=0}^{k} d_i > \sum_{i=1}^{k} c_i.$$

Proof. Assume without loss of generality that $c_1 > 1$. We claim that the two-layered design $\{d_0, d_1, \ldots, d_k\}$ where $d_0 = 1$ and $d_i = c_i$ for all $i = 1, \ldots, k$ is still M-bounded and, of course,

$$\sum_{i=0}^{k} d_i > \sum_{i=1}^{k} c_i.$$

Consider the conditions of Lemma 7.3. Adding d_0 to the original design clearly doesn't violate the first restriction. Regarding the second condition, we must only verify that, for all $j = 1, \ldots, k$, $d_0 + d_j + (j-1) \le M$. This immediately holds whenever $j > 1$, since

$$\begin{aligned} d_0 + d_j + (j-1) &= 1 + c_j + (j-1) \\ &= c_1 - (c_1 - 1) + c_j + (j-1) \\ &\le c_1 + c_j + (j-2) \le M. \end{aligned}$$

As for $j = 1$, since $k > 1$ and $c_1 + c_2 \le M$, it must follow that $c_1 < M$, and thus $1 + c_1 = 1 + d_1 \le M$. □

We can now prove the upper bound on $b(M)$.

Proposition 7.8.

$$b(M) = \begin{cases} (\frac{M+1}{2})^2 & \textit{if } M \textit{ is odd,} \\ \frac{M}{2}(\frac{M}{2}+1) & \textit{if } M \textit{ is even.} \end{cases}$$

Proof. Let $\{c_1, \ldots, c_k\}$ be an M-bounded two-layer design. Note the following:

- k cannot exceed M. Otherwise, if $k > M$,
$$c_1 + c_k + (k-2) \geq 1 + 1 + (k-2) = k > M$$
and the design is not M-bounded.
- Whenever $k = M$, Lemma 7.6 has already shown that $\sum_{i=1}^{M} c_i$ cannot exceed the claimed bound on $b(M)$.

We are therefore left with the case $k < M$. First, note that we can exclude the case where $k = 1$, since $c_1 \leq M$ and Proposition 7.5 already implies that $b(M) \geq M$ for all M.

Thus, we examine the case where $2 \leq k < M$, and we assume by way of contradiction that the given design is optimal, i.e., $b(M) = \sum_{i=1}^{k} c_i$. We now prove by induction that, for all $i = 1, \ldots, \lceil \frac{k}{2} \rceil$, $c_i = c_{k+1-i} = i$.

Lemma 7.7 provides us with the base of the induction, since if the design is optimal then necessarily $c_1 = c_k = 1$. Thus, assume that, for any j in the range $2, \ldots, \lceil \frac{k}{2} \rceil$, for all $i < j$ it holds that $c_i = c_{k+i-1} = i$, and consider c_j (the proof for c_{k+1-j} is symmetric). If $c_j \neq j$, then one of the following three cases holds:

Case I. ($c_j \leq j - 1$.) Define the two-layered design $\{d_1, \ldots, d_k\}$ where, for all $i \neq j$, $d_i = c_i$ and $d_j = c_j + 1$. Showing that $\{d_1, \ldots, d_k\}$ is M-bounded would contradict the optimality of $\{c_1, \ldots, c_k\}$. To this effect, we note that for all $i < j$

$$\begin{aligned} d_i + d_j + (j-i-1) &\leq i + j + (j-i-1) \\ &= 2j - 1 \leq k < M, \end{aligned}$$

whereas for all $t > j$

$$\begin{aligned} d_t + d_j + (t-j-1) &\leq c_t + j + (t-j-1) \\ &= c_t + (t-1) \\ &= c_t + c_1 + (t-2) \leq M. \end{aligned}$$

Case 2. ($j < c_j \le M - j$.) Define the two-layered design $\{d_1, \dots, d_k\}$ where, for all $i \neq (j-1)$, $d_i = c_i$ and $d_{j-1} = c_{j-1} + 1 = j$. We again show that $\{d_1, \dots, d_k\}$ is M-bounded, since for all $i < j-1$

$$\begin{aligned} d_i + d_{j-1} + (j-i-2) &= i + j + (j-i-2) \\ &< i + j + (j-i-1) \\ &< c_i + c_j + (j-i-1) \le M. \end{aligned}$$

Also,

$$d_j + d_{j-1} \le M - j + j = M,$$

and for all $t > j$

$$\begin{aligned} d_t + d_{j-1} + (t-j) &= c_t + j + (t-j) \\ &= c_t + (j+1) + (t-j-1) \\ &\le c_t + c_j + (t-j-1) \le M. \end{aligned}$$

Case 3. ($j < c_j = M - (j-1)$.) For this case, we first prove that $k = 2j - 1$:

- $k > 2j - 2$, since j does not exceed $\lceil \frac{k}{2} \rceil$ during the induction, and so $j < \frac{k+2}{2}$.
- $k \le 2j - 1$, since the design is M-bounded, and so

  $$0 \ge [c_j + c_k + (k-j-1)] - M = [M-(j-1)+1+(k-j-1)] - M = k - 2j + 1.$$

Since $k < M$, $k = 2j - 1$ implies that $M \ge 2j$, or equivalently $\frac{M-2}{2} \ge j - 1$. Therefore,

$$\begin{aligned} \sum_{i=1}^{k} c_i &= c_j + \sum_{i=1}^{j-1} c_i + \sum_{i=j+1}^{k} c_i \\ &= c_j + \sum_{i=1}^{j-1} (c_i + c_{k+1-i}) \\ &= M - (j-1) + 2\sum_{i=1}^{j-1} c_i \quad \text{(by the induction hypothesis)} \\ &= M - (j-1) + j(j-1) \\ &= M + (j-1)^2 \\ &\le M + \left(\frac{M-2}{2}\right)^2 = 1 + \left(\frac{M}{2}\right)^2 < b(M) \ \ \forall M > 2. \end{aligned}$$

Having ruled out all three cases above, we conclude that c_j (and also c_{k+1-j}) must equal j, completing the induction. Finally, since $c_i = c_{k+i-1} = i$, $i = 1, \dots \lceil \frac{k}{2} \rceil$ and $k < M$, $\sum_{i=1}^{k} c_i < b(M)$, which completes the proof. $\square$

Comparing the result of Proposition 7.8 to the substitution of $L = 1$ in Corollary 6.2, we deduce that by moving from fixed clustering to variable clustering within layer 1, while keeping M fixed, one can double b—thus halving the amount of worst-case filtering required by any query.

Corollary 7.9. *For a given value of b, when variable-cluster designs are considered, $M_{min}(1, b) = 2\sqrt{b} + \mathcal{O}(1)$.*

Comparing this result to the substitution of $L = 1$ in Equation (6.3), we deduce that by moving from fixed clustering to variable clustering within layer 1, while keeping b fixed, one can reduce M_{min} by a factor of $\sqrt{2}$.

8. Experimental Results

In this section we experiment with the benefits and trade-offs of the (L, c) canopy design. We have implemented this design in Trevi, an intranet search engine built in IBM that is currently used to serve queries in the IBM world-wide intranet [Fontoura et al. 04]. We ran all experiments on a Linux system with a dual 2.4GHz Intel Xeon processor and 4 GB of main memory. In order to account for the I/O overhead in processing the queries, we used cold buffers in all experiments that measure running time. We used real data sets from the IBM intranet, augmented with two synthetically generated numeric fields per document. The values for these fields were derived from two random distributions on real numbers, causing the number of distinct values per field to be very close to the number of documents in the dataset. Furthermore, the distributions were derived independently of the documents' content and of their docIDs, so the numeric values are independent of any other postings list in the index and are uncorrelated to the order of the documents in the inverted index. For all experiments we used an index of 2.5 million documents. The index size for the IBM intranet is approximately 5 million documents after duplicate elimination. However, numeric postings lists of 2.5 million documents in size are common, since not all documents have numeric fields.

$\lvert L\rvert$	Theoretical c	Used c
2	70.7	64
3	17.1	16
4	8.40	8
5	5.49	4

Table 1. Index configurations tested.

8.1. Index Size and Build Performance

For measuring index size and build performance, we built several configurations of the numeric postings, varying the number of extra layers (L) and the clustering factor (c). Let us define $|L| = L + 1$ to be the total number of layers. For these experiments we fixed the filtering factor F by setting the block sizes for layer 0 to 4,096 bytes. In this configuration, each block in layer 0 can hold approximately 250 values, which caused the number of blocks in layer 0 to be 10,000 for our 2.5 million document index. We then varied $|L|$ and c, computing the best possible c for each $|L|$ using the result of Proposition 1. The index configurations we used are summarized in Table 1. In our implementation the value of c must be a power of two, so we present two values for c in Table 1: theoretical c is the value computed from Proposition 6.1, and used c is its closest power of two, which is the value we used for the experiments. Besides the configurations listed in Table 1, we have also built indexes with no extra layers ($|L| = 1$), for which the clustering factor is irrelevant.

Table 2 shows the sizes of the numeric postings lists for each index. It also shows the index-space overhead due to these postings, i.e., the percent of space used by the numeric postings lists in the index. This overhead is heavily dependent on the number of numeric fields per document, and it grows linearly with this number (as mentioned above, we indexed two numeric fields per document). An important point is that the overhead due to layer 0 ($|L| = 1$) is higher than the overhead due to the extra layers since the numeric value itself is only stored in the layer-0 postings lists, as discussed in Section 6.4.

$\lvert L\rvert$	Size (MB)	Overhead
1	54	0.45%
2	72	0.60%
3	90	0.75%
4	108	0.90%
5	126	1.05%

Table 2. Numeric postings list sizes for the different index configurations.

$\|L\|$	Time (ms)	Overhead
1	27123	1.01%
2	51857	1.93%
3	75090	2.83%
4	100934	3.77%
5	126028	4.70%

Table 3. Numeric postings list build time for the different index configurations.

Table 3 shows the performance for building the numeric postings lists. It also shows the time overhead (percent of indexing time) for building the numeric postings lists for the extra layers. Our implementation used the RAM-limited approach where we first sorted the numeric postings by value and wrote them to disk. Then, each layer was built by scanning that sorted list, once per layer. This overhead reached up to 4.70%. As expected, both the index size and the build time grow linearly with the number of layers. Nevertheless, the overall impact of the extra layers is small, both in size and in build time performance.

8.2. Runtime Performance of Range Queries

Figure 7 plots the query response times as a function of the selectivity of the numeric field. Selectivity pertains to the fraction of the 2.5 million documents that satisfy the numeric constraint. We varied the selectivity of the numeric field from 1 down to 2^{-i}, for $i \in \{1, \ldots, 10\}$. The y-axis is plotted in logarithmic scale. Most of the plots correspond to the number of numeric postings layers, with the relatively level line plotting the performance resulting from using the filtered postings list mechanism described in Section 4. We ran these experiments with cold buffers to avoid any caching. The bulk of the time for all the runtime numbers is due to I/O: the CPUs were mostly idle for every pair of query and index configuration that we tested. The approach based on no extra layers ($|L| = 1$) was not considered, since it turned out to be extremely inefficient due to the large merge overhead. Therefore, we omit the results for $|L| = 1$ in Figure 7.

The filtered-postings approach behaved consistently across the entire selectivity spectrum, timing at about 8,000 ms per query, clearly worse than any of the multilayered designs. The main reason for this higher runtime overhead is the extra I/O required, since each posting entry in the filtered list needs to store the value in its payload (8 bytes per posting entry), which is not required for the postings lists for the extra layers. Moreover, CPU utilization is also higher since it needs to examine the data payload for every posting entry, checking if it is within the specified range.

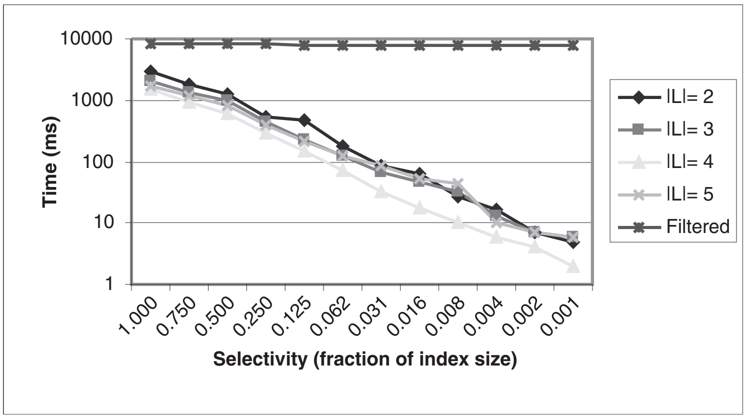

Figure 7. Query response times versus query selectivity for several values of $|L|$.

Among the layered schemes, query throughput increases with the number of extra layers until query selectivity reaches the 0.8% point. For the more selective queries, all layered designs attain comparable performance figures (less than 10 ms per query). This is expected since, for larger indexes, reducing the merge overhead is a key performance factor for nonselective queries. Figure 7 also shows that increasing the number of layers to more than four did not improve performance for this index size. The reason is that for $|L| = 4$ the number of postings lists to merge is already small. In this case the extra layers do not add significant benefit, and for very selective queries they even add some noticeable overhead.

8.3. Runtime Performance of Range Queries Combined with Text Terms

The previous experiments illustrated scenarios in which the numeric constraint drove the query execution. In this section we study the case where a more selective text predicate drives the execution. The bulk of the query evaluation time is due to I/O on the postings lists for each of the query terms. In the case where one or more text terms drive the query execution, we are interested in the overhead that is added by the numeric postings lists.

In Figure 8 we ran experiments in our 2.5 million document index built with $|L| = 4$. We used the keyword "almaden" for the text part of the query since it is selective, appearing in only 1.6% of the documents in the index. We varied the selectivity of the numeric field from 1 down to 2^{-i}, for $i \in \{1, \ldots, 10\}$. The y-axis is plotted in logarithmic scale. This graph also shows the running time of a query involving two text terms: "almaden" and a synthetic term that appears in every document. This flat curve is labeled "Text." When the selectivity is 1,

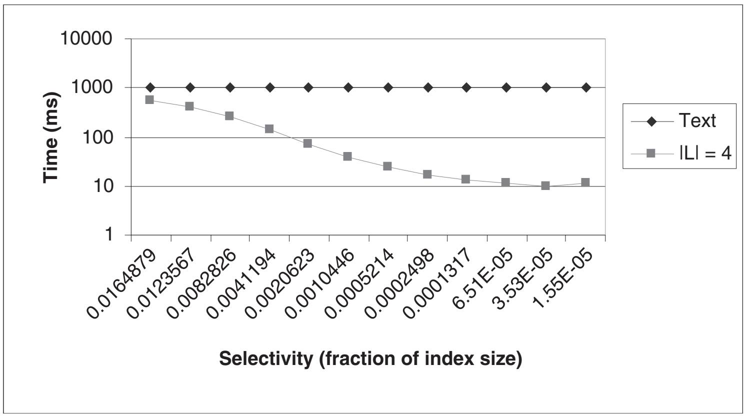

Figure 8. Query performance for a range query combined with keyword "almaden."

i.e., the furthest point to the left, both queries ("Text" and $|L| = 4$) produce the same number of results. The comparison between these two queries shows that the overhead for the numeric list is lower than the overhead for a text term: the curve for "Text" is always above the curve of $|L| = 4$, even when the selectivity is 1. This is due to the fact that the numeric lists for the extra layers are smaller than postings lists for regular text terms, since they do not store any payload information.

The running time for the query involving the numeric field ($|L| = 4$) decreases proportionally to the number of results. This is because the combined query is able to run more efficiently since it can process fewer documents by performing

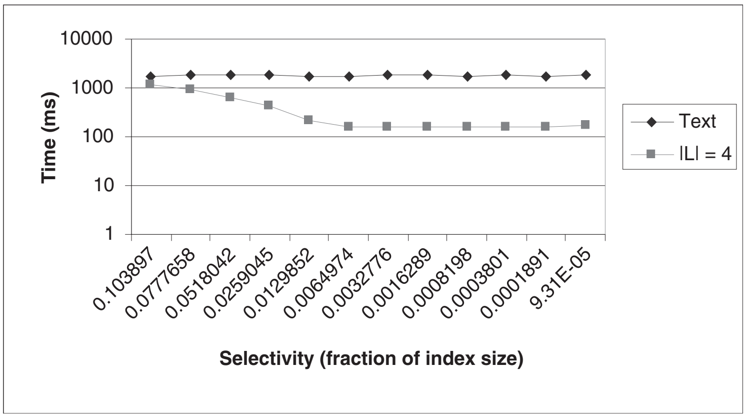

Figure 9. Query performance for a range query combined with keyword "java."

efficient cursor movements (larger skips) on the postings lists, saving considerable amounts of I/O. When the numeric term is less selective, less skipping is possible. In Figure 9 we used the keyword "java" instead of "almaden." The selectivity of "java" is higher: it appears in 10.3% of the documents. Nevertheless, the results are still similar to those presented in Figure 8.

9. Conclusions and Future Work

This paper studied the problem of efficiently supporting numeric search in search engines. We began by discussing the shortcomings of several naive approaches to this problem. We then proposed several modifications and extensions to a state-of-the-art algorithm for the problem, addressing implementation issues along the way. Our ensuing analysis described how inverted lists representing numeric fields can be constructed so as to maximize query-processing performance while respecting limits on index size and build time, or conversely, how index space and build time can be minimized while maintaining guarantees on runtime performance. Our experimental results confirmed our performance analysis and demonstrated significant performance improvement (on the order of 10 times) over the naive filtered posting list approach.

For future work, we plan to further investigate multilayered schemes with variable clustering, going beyond the case of $L = 1$ studied in Section 7. For example, for $b = 42$ (42 blocks in layer 0), Figure 10 shows (half of) a design with three extra layers where no more than eight merges are needed for any query. Note that L_0 and L_2 constitute an optimal 12-bounded two-layer design as proven in Section 7. This is an improvement over optimal (L, c)-canopies—as shown in Section 6.3, with three extra layers, $M_{min}(3, 42) = 11$—barely better than the optimal 12-bounded two-layer design.

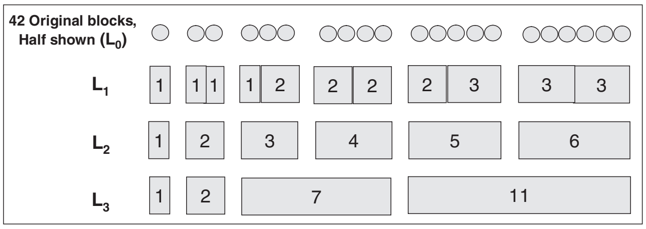

Figure 10. Left half of a multilayered variable clustering design with $b = 42$ and three extra layers.

We also plan to integrate our approach in XML search engines and to study how to support XPath/XQuery queries with parametric restrictions in inverted index-based search engines. Note that Holistic twig-join [Bruno et al. 02], which is the current state-of-the-art in XML query-processing algorithms over inverted indexes, efficiently checks the structural constraints of XML while traversing the postings lists. However, it does not check the parametric or numeric restrictions of the query. We intend to extend the holistic twig-join algorithm to check numeric restrictions using the approach proposed in this paper.

Acknowledgments. We would like to thank Kevin Beyer, Chavdar Botev, Andrei Broder, Volker Markl, Andreas Neumann, and Eugene Shekita for the many helpful comments on our design and on our early drafts. Work for this paper was done while the first and fourth authors were at the IBM Almaden Research Center and while the third author was at the IBM Silicon Valley Lab.

References

[Agrawal et al. 02] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. "DBXplorer: Enabling Keyword Search Over Relational Databases." In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, p. 627. New York: ACM Press, 2002.

[Arasu et al. 01] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. "Searching the Web." *ACM Transactions on Internet Technology* 1:1 (2001), 2–43.

[Baeza-Yates and Ribeiro-Neto 99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Reading, MA: Addison Wesley, 1999.

[Bhalotia et al. 02] Gaurav Bhalotia, Charuta Nakhe, Arvind Hulgeri, Soumen Chakrabarti, and S. Sudarshan. "Keyword Searching and Browsing in Databases Using BANKS." In *Proceedings of the 18th International Conference on Data Engineering*, pp. 431–440. Los Alamitos, CA: IEEE Computer Society, 2002.

[Brin and Page 98] Sergey Brin and Lawrence Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine." *Computer Networks (Proceedings of the 7th International World Wide Web Conference)* 30:1–7 (1998), 107–117.

[Broder et al. 03] Andrei Broder, David Carmel, Miki Herscovichi, Aya Soffer, and Jason Zien. "Efficient Query Evaluation Using a Two-Level Retrieval Process." In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, pp. 426–434. New York: ACM Press, 2003.

[Bruno et al. 02] N. Bruno, N. Koudas, and D. Srivastava. "Holistic Twig Joins: Optimal XML Pattern Matching." In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pp. 310–321. New York: ACM Press, 2002.

[Burrows 96] Michael Burrows. "Object-Oriented Interface for an Index." US patent 5809502, 1996.

[Carmel et al. 03] D. Carmel, Y. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer. "Searching XML Documents via XML Fragments." In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 151–158. New York: ACM Press, 2003.

[Cutting and Pedersen 90] Doug Cutting and Jan Pedersen. "Optimizations for Dynamic Inverted Index Maintenance." In *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 405–411. New York: ACM Press, 1990.

[Fontoura et al. 04] Marcus Fontoura, Eugene Shekita, Jason Zien, Sridhar Rajagopalan, and Andreas Neumann. "High Performance Index Build Algorithms for Intranet Search Engines." In *Proceedings of the 30th International Conference on Very Large Data Bases*, pp. 1158–1169. San Francisco: Morgan Kaufmann, 2004.

[Gravano 01] Luis Gravano, editor. *IEEE Data Engineering Bulletin (Special Issue on Text and Databases)* 24:4, 2001.

[Gravano et al. 03] Luis Gravano, Panagiotis G. Ipeirotis, Nick Koudas, and Divesh Srivastava. "Text Joins in an RDBMS for Web Data Integration." In *Proceedings of the 12th International World Wide Web Conference*, pp. 90–101. New York: ACM Press, 2003.

[Guttman 84] Antonin Guttman. "R-trees: A Dynamic Index Structure for Spatial Searching." In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pp. 47–57. New York: ACM Press, 1984.

[Heinz and Zobel 03] Steffen Heinz and Justin Zobel. "Efficient Single-Pass Index Construction for Text Databases." *JASIST* 54:8 (2003), 713–729.

[Hristidis and Papakonstantinou 02] V. Hristidis and Y. Papakonstantinou. "DISCOVER: Keyword Search in Relational Databases." In *Proceedings of the 28th International Conference on Very Large Data Bases*, pp. 670–681. San Francisco: Morgan Kauffman, 2002.

[Kaushik et al. 04] R. Kaushik, R. Krishnamurthy, J. Naughton, and R. Ramakrishnan. "On the Integration of Structure Indexes and Inverted Lists." In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pp. 779–790. New York: ACM Press, 2004.

[Melnik et al. 01] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. "Building a Distributed Full-Text Index for the Web." In *Proceedings of the 10th International Conference on World Wide Web*, pp. 396–406. New York: ACM Press, 2001.

[Press et al. 88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge, UK: Cambridge University Press, 1988.

[Raghavan and Garcia-Molina 01] Sriram Raghavan and Hector Garcia-Molina. "Integrating Diverse Information Management Systems: A Brief Survey." *IEEE Data Engineering Bulletin* 24:4 (2001), 44–52.

[Raghavan and Garcia-Molina 03] Sriram Raghavan and Hector Garcia-Molina. "Complex Queries over Web Repositories." In *Proceedings of the 29th International Conference on Very Large Data Bases*, pp. 33–44. San Francisco: Morgan Kaufmann, 2003.

[Ramsak et al. 00] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. "Integrating the UB-Tree into a Database System Kernel." In *Proceedings of the 26th International Conference on Very Large Data Bases*, pp. 263–272. San Francisco: Morgan Kaufmann, 2000.

[Samet 90] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Reading, MA: Addison-Wesley, 1990.

[Spertus and Stein 00] Ellen Spertus and Lynn Andrea Stein. "Squeal: A Structured Query Language for the Web." *Computer Networks (Proceedings of the 9th International Conference on World Wide Web)* 33:1–6 (2000), 95–103.

[Witten et al. 99] Ian Witten, Alistair Moffat, and Timoty Bell. *Managing Gigabytes*, Second edition. San Francisco: Morgan Kaufmann, 1999.

Marcus Fontoura, Yahoo! Research, 2821 Mission College Blvd., Santa Clara, CA 95054 (marcusf@yahoo-inc.com)

Ronny Lempel, IBM Haifa Labs, Mount Carmel Campus, Haifa 31905, Israel (rlempel@il.ibm.com)

Runping Qi, 7588 Barnhart Place, Cupertino, CA 95014 (runping@yahoo-inc.com)

Jason Zien, 928 Wright Ave., #408, Mountain View, CA 94043 (jasonz@alumni.cse.ucsc.edu)

Received September 15, 2005; accepted June 12, 2006.