RJ10329 (A0410-003) October 5, 2004
Computer Science

IBM Research Report

Inverted Index Support for Parametric Search

Marcus Fontoura, Jason Zien
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Ronny Lempel
IBM Research Laboratory
Mount Carmel Campus
Haifa 31905, Israel

Runping Qi
IBM Silicon Valley Laboratory
555 Bailey Avenue
San Jose, CA 95141

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Inverted Index Support for Parametric Search

Marcus Fontoura
Jason Zien
IBM Almaden Research Ctr.
650 Harry Road
San Jose, CA 95120, USA
fontoura@us.ibm.com
jasonz@us.ibm.com

Ronny Lempel
IBM Research Lab
Mount Carmel Campus
Haifa 31905, Israel
rlempel@il.ibm.com

Runping Qi
IBM Silicon Valley Lab
555 Bailey Avenue
San Jose, CA 95141, USA
runping@us.ibm.com

ABSTRACT

As an attempt to integrate data from the Web and corporate knowledge portals with data residing in proprietary databases, search engines are required to support a more complex set of features, such as Boolean, XML, and parametric restrictions. In this paper we propose a scheme by which an inverted-index based search engine can efficiently support queries that contain parametric restrictions in addition to standard, free-text portions. We show how inverted lists for parametric fields can be constructed and used seamlessly during query evaluation time. We will 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. We concisely present the tradeoff between index size, build time, and runtime performance. Finally, our experimental evaluation shows significant performance benefits when compared to alternative approaches.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Information Search and Retrieval,Systems and Software*

General Terms

Algorithms, Performance

Keywords

Parametric search, search engines, query evaluation

1. INTRODUCTION

We are presently witnessing increasing attempts to narrow the gap between search and retrieval capabilities over unstructured, semi-structured, and structured data [4, 5, 8, 9]. Databases, traditionally used to manage structured data, are increasingly developing free-text search capabilities to efficiently and effectively handle queries over free-text columns of information in their records [4]. Free-text retrieval systems, such as search engines, now commonly exhibit some functionality of searching semi-structured data via fielded-search mechanism. In this paper, we continue this trend by augmenting free-text search engines with the ability of efficiently handling parametric clauses - clauses that restrict the allowed range of values for numeric fields that might be associated with documents in the corpus. Specifically, we are targeting corpora where free-text documents have numeric meta-data fields associated with them (e.g., date of creation, prices of products, etc).

Queries over such corpora are composed of a free-text part, followed by one or more parametric constraints. Concretely, a parametric constraint consists of an allowed range of values per one of the numeric meta-data fields. Parametric constraints can be one-sided, defining ranges ($[v, \infty]$ or $[-\infty, v]$), or two sided, defining ranges $[v_{min}, v_{max}]$. Two-sided queries are more general and so most of the discussion will center on those. Parametric terms are treated as boolean restrictions - we are not attempting to rank documents by their parametric values. Documents that respect the parametric constraints will be ranked according to the free-text portion of the query.

The dominant query evaluation model for Web and Intranet search engines is the document-at-a-time (D@T) [1, 2]. In this model, posting lists of all query terms are traversed in parallel during query execution. Since the evaluation model for the parametric constraints should flow well within the existing query engine, it is best if it is based on (or can be encapsulated by) posting lists.

Our Contribution

We propose a design where the values corresponding to each parameteric field are indexed by constructing multiple *layers* of posting lists, where the aggregation of each layer's lists roughly contains a posting element for each occurence of the parametric field in the corpus. It follows that the space (and build time) required to index the parametric values is linear in the number of layers. Naturally, each layer organizes the distribution of the parametric data across the documents differently. We explain how to efficiently construct these posting lists, taking into account also the amount of available RAM during the index build phase.

The multiple representations of the data in the index lay-

ers is what allows the index to efficiently evaluate queries during runtime. Support for each parametric constraint is realized by intersecting some of the posting lists in the layers described above with other posting lists as determined by the free-text, non-parametric portion of the query. The cost of query evaluation will be governed by the number of parametric posting lists that must be considered per restriction (denoted by M), in addition to some filtering that will need to be done on F entries of (at most two) parametric posting lists.

We will show how M and F can be lowered, at the cost of increasing the number of index layers, L. Furthermore, we concisely present the tradeoff between F, L and M - given tolerable values of any pair of these parameters we will show how to minimize the third. Our analysis of this tradeoff results in performance guarantees that are completely insensitive to the type of parametric values (integer/floating point, positive/negative), to the distribution of values across the documents (clustered/non clustered values), and to the granularity, or precision, requested by the queries. We thus make no assumptions on any of these issues.

To ease the presentation, the analysis will mostly focus on the case where for each parametric attribute, 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 and the number of parametric document-value pairs to be indexed. However, the indexing and retrieval algorithms we present are easily extended to support multiple values that may be associated with a document. Section 4.3 elaborates on the multi-valued case.

The rest of this paper is organized as follows. Section 2 discusses naive solutions to the parametric search problem. Section 3 surveys related work. Section 4 presents our proposed data structure of parametric posting lists, analyses performance aspects and discusses implementation issues. Section 5 reports on experiments we conducted with an implementation of our scheme inside an intranet search engine. Section 6 addresses some theoretic issues that are tangent to our solution. We conclude in Section 7.

2. NAIVE SOLUTIONS

Perhaps the most straightforward incorporation of parametric data in an inverted index is to create a single posting list for each parametric field. Like all posting lists, the entries in the parametric list will be ordered by document IDs (docIDs), with each posting element storing the value that corresponds to the document. During evaluation, the parametric posting list is used as a filter: essentially, evaluation is driven by the non-parametric terms, and candidate documents are filtered by their values. We refer to this posting list as a *filtered* posting list. This could be reasonably efficient when the non-parametric part of the query is selective (i.e., relatively few candidate results are presented to the parametric filter), but is quite inefficient if the selectivity lies mostly in the parametric 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 parametic posting elements that must be scanned, regardless of the eventual size of the result set for the query.

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 parametric restriction, thus identifying the documents that match the parametric 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, that are typically produced in docID order. This is not very efficient in cases where the parametric restriction was not highly selective. Furthermore, this join-based algorithm does not fit well within the architecture and flow of search engines.

A more advanced solution, which is efficient whenever the number of distinct values per parameter is small, is to have a posting list (sorted, naturally, by docID) per value. This solution also requires a sorted table of all distinct values, with each entry of the table pointing to the corresponding posting list (in other words, a lexicon for this parameter, sorted by values). During query evaluation, one consults the lexicon to determine the indexed values that fall within the requested range. Then, the corresponding posting lists are OR-ed (merged by docID) and AND-ed with the rest of the query's terms. This approach is not efficient for general value distributions, however: as the number of distinct values grows, the lexicon's size grows, as well as the number of (now very small) posting lists that need to be ORed during evaluation. In many cases, accessing each posting list requires some disk I/O, and so performance may rapidly deteriorate as more distinct posting lists are needed.

While the above three methods all suffer from inefficiencies in runtime performance, note that they index each (docID, value) pair once. Thus, the overhead in terms of index space is kept minimal. Section 4 will show how runtime performance can be improved at the expense of additional index space.

3. RELATED WORK

TODO - Marcus?

4. PARAMETRIC POSTING LISTS

This section presents our proposed scheme for supporting parametric queries within a posting-list driven search engine. We explain how to build parametric posting lists, and how to use them when evaluating queries that contain parametric range constraints. We analyze the time and space requirements of the index build phase, derive bounds on runtime performance given the parameters of the index, and analyze the tradeoff between runtime performance and index build complexity.

Section 4.1 presents the first step toward our solution, which is then fully explained and analyzed in Section 4.2. We conclude in Section 4.3 with a discussion of several implementation issues that arise from our scheme.

4.1 A First Step: Equal-Sized Blocks of Increasing Value Ranges

As discussed in Section 2, when the number of distinct values present for some parametric field is large (when few documents share the same value), keeping a posting list for each value results in bad performance. A query may require us to fetch many short posting lists, and to then 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 propose a scheme that creates *longer and fewer* posting lists by having each list ℓ cover a *range* of values, $r(\ell)$.

Specifically, we propose the following 3-step process that creates b posting lists, each of size $\frac{N}{b}$, and two b-sized lookup tables T_{min} and T_{max}.

1. Sort all N ⟨docID,value⟩ 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.
3. Create a posting list from each of the b blocks, by resorting them by docIDs. Note that while each block is 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, $\mathcal{O}(N \log N)$. The index space complexity is $\mathcal{O}(N)$ - still minimal.

During runtime, upon receiving a query that requests a value range of $R = [v_{min}, v_{max}]$, evaluation proceeds as follows:

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

$$i_{min} = \max_{j \in \{0,\ldots,b-1\}} \{ j : T_{max}[j] < v_{min} \} + 1$$
$$i_{max} = \min_{j \in \{0,\ldots,b-1\}} \{ j : T_{min}[j] > v_{max} \} - 1$$

   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 posting lists i_{min} through i_{max} includes all indexed values that fall within the range R. Formally,

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

2. Construct a *filtered* posting list from $\ell_{i_{min}}$ that only considers entries whose values are no smaller than v_{min}. Similarly, construct a filtered posting list from $\ell_{i_{max}}$ that only consider entries whose values are no greater than v_{max}. Denote these filtered lists by $f_{i_{min}}, f_{i_{max}}$ respectively [1].
3. Merge the following posting 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 posting list to the evaluation.

The above design 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.

[1] Whenever $i_{min} = i_{max}$, the single posting list $\ell_{i_{min}}$ is filtered by both the lower bound v_{min} and the upper bound v_{max}.

Clearly, we have a tradeoff 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 next.

4.2 L-Layered c-Regular Super-Block Design

This section shows how one can extend the b equal-sized posting lists of the previous section to reduce the runtime demands of parametric queries, at the cost of increasing index size and build time. Specifically, this design will add several additional *layers* of posting lists, where each layer is of size N. We term the b postings of size $\frac{N}{b}$ as *layer*-0.

Let $c > 1, L \geq 1$ be natural numbers. L will denote the number of *additional* layers of posting lists for the parameter, and c will denote a *clustering* factor. For simplicity of exposition, we assume in what follows that c^L divides b.

The basic idea is for each layer j, $j = 1, \ldots, L$, we construct $\frac{b}{c^j}$ posting lists of size $\frac{Nc^j}{b}$. As in layer 0, each document appears in exactly one posting list of each layer. Posting list i, $i = 0, \ldots, \frac{b}{c^j} - 1$ of layer j is composed of the merging (by docID) of posting lists $ic, \ldots, (i+1)c - 1$ of layer $j - 1$. In other words, each layer clusters the posting lists of the previous layer in groups of size c, ending up with larger (but fewer) posting lists. We call this design an L-layered c-regular super-block design. Figure 1 shows a regular super-block design with 3 extra layers and clustering factor 2.

4.2.1 Query Evaluation at Runtime

During runtime, the following steps are executed:

- Consult T_{min}, T_{max} as explained in the previous subsection. This implies which layer-0 blocks define the range.
- Build filtered postings for the two extreme layer-0 postings.
- Select the appropriate postings from each layer so as to minimize the number of selected postings. We label the blocks using a special numbering scheme in which each block in level 0 is numbered sequentially and each higher level block is labeled using its level 0 equivalent starting point (Figure 1). The method we use to choose the blocks is a greedy algorithm which iteratively picks the largest next block which fits (see Figures 2 and 3).
- Merge the selected posting lists, OR-ing them via heap merge. The compexity os this step is $\mathcal{O}(\sum_{i=1}^{m} |\ell_i| \log m)$, where the m lists $\ell_1, \ldots, \ell_m$ are OR-ed and $|\ell_i|$ denotes the length of list i.

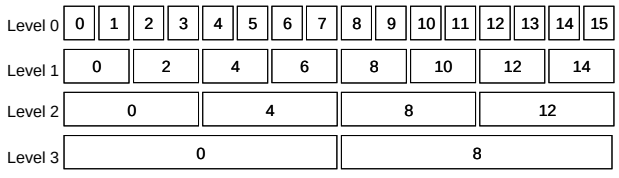

Figure 1: Numbering of blocks for Block Selection Algorithm

```
void selectBlocks(
    int m,          // minimal level-0 block index
    double v_min,   // the minimal value desired
    int M,          // maximal level-0 block 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==M should be
    // handled separately
    b = getBlock(m,0);
    if ( v_min > getMinValueInBlock(b) ) {
        OrPosting.addFiltered(b);
        from=from+1;
    }
    b = getBlock(M,0);
    if ( v_max < getMaxValueInBlock(b) ) {
        OrPosting.addFiltered(b);
        to=to-1;
    }

    // blocks from...to should all be taken
    // as is (no filtering)
    while (from ≤ to) {
        level = dive(from, to, clustering);
        OrPostings.add(getBlock(from,level));
        from += clustering^level;
    }
}
```

Figure 2: Block Selection Algorithm

```
int dive(
    int fromBlockNumber,
    int toBlockNumber,
    int clustering )
{
    int level = 0;
    int range = toBlockNumber - fromBlockNumber + 1;
    // check if the level+1 is a good candidate
    while ( (clustering^(level+1) divides fromBlockNumber )
        && (range ≥ clustering^(level+1))
        && ((level+1) < MaxLevels) ) {
        level++;
    }
    return level;
}
```

Figure 3: Helper Function for Block Selection

Figure 4 exemplifies the posting lists that are OR-ed for a certain query that ranges from blocks 2 to 14, in a scheme with 2 extra layers and clustering factor 2. In this example only 6 posting lists are accessed, as oppoesed to 12 in the scheme that does not use extra layers.

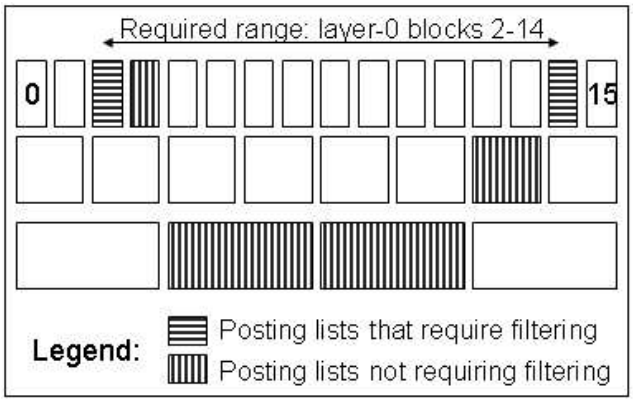

Figure 4: Posting list selection during query evaluation

Worst-case performance guarantees

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:

1. 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 of layer 0. Each such posting is of size $\frac{N}{b}$.

2. The number of postings that should be OR-ed for this parameteric 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 $2c$ postings (c postings with low values, c postings with high values). From each level $j=1,\ldots,L-1$, we need to consider no more than $2(c-1)$ postings. In level L, we may need to OR all but two of the postings: $\frac{b}{c^L}-2$. 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 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 1. *The optimal value of c in an L-layered regular super-block design, 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 block in layer-0 should contain no more than $\frac{F}{2}$ documents. This implies

that the number of blocks in layer-0 is $b = \frac{N}{2F}$. [2] 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 deduce that

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

So far we have shown how to minimize M for tolerated values of b (or, equivalently, F) and L. Equation 1 readily allows us to maximize b (thereby minimizing F) given tolerated values of L and M:

COROLLARY 1. *The maximal value of b in an L-layered regular super-block design where up to M OR-ed postings are tolerated is $b_{max}(L,M) = 2[\frac{M+2L}{2L+2}]^{L+1}$.*

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 [7].

So far we have considered the clustering factor c to be the same across all layers. At first glance, it may seem that allowing each layer to have its own clustering factor could 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? The answer turns out to be 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 posting lists to be OR-ed when using an L-layered design with clustering coefficients $c_1, \ldots, c_L$.

PROPOSITION 2. *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 similar considerations as used above, we bound the number of postings 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 the above 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 c_i's should be equal.

[2] For simplicity's sake, assume that F divides N.

4.2.2 Index build time

We now examine how L-layered c-regular super-block designs 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 we can fit $2N$ records in memory, giving us the ability to hold two layers simultaneously in memory. As explained in Section 4.1, creating layer-0 requires $\mathcal{O}(N \log N)$ time and is dominated by the initial sorting of the N ⟨docID,value⟩ pairs by value. Layer-0 can then be written contigiously 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 blocks of layer j at a time. Layer $j+1$ is then written to disk in a contiguous manner, one block 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 ⟨docID,value⟩ pairs sorted by docID.
2. We sort the N pairs by value into another N-sized buffer, and build the tables T_{min} and T_{max}.
3. Reusing the second N-sized buffer, we consult the T-tables and write, for every document d, the index of the layer-0 block 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 $\frac{b}{c^j}$ blocks of layer j. All the blocks 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 block corresponding to d, which we wrote in the second buffer. Let k denote that block number. The pair $\langle d, v \rangle$ belongs in block $\lfloor \frac{k}{c^j} \rfloor$ of layer j, and we write it to the first empty space in that block. Since we scan the data by docIDs, we are guaranteed that each layer-j block 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 block 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.

4.3 Final Implementation Notes

Removal of values in posting lists of levels other than 0

The postings of layer 0 are composed of ⟨docID, value⟩ ordered pairs. This allows for these posting lists to be filtered by value when some of the values in the list fall outside the parametric range R defined by the query. Postings of

levels deeper than 0, however, are never filtered - they only participate in the evaluation when R fully contains the parametric range of the block. Therefore, the ⟨value⟩ component is redundant for the sake of retrieval and can be dropped, resulting in a more compact posting list representation, where each element carries no payload beyond the docID.

Multiple values per document

In some cases, documents may pertain to multiple values of the parameter in question. For example, a page from a product catalog might describe products of different prices or 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 posting lists. However, in all other layers, multiple occurences of the same docID in a block 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 levels (the values are redundant there).

*Value-based clustering of layer-*0

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

```
// B - the current block being constructed
// V - the set of distinct values
// B_v - the set of documents associated with value v
B ← φ
Foreach v ∈ V (ordered by increasing values){
    if (|B| + |B_v| ≤ F):
        B ← B ∪ B_v
    else {
        write B (sorted by docIDs)
        B ← B_v
    }
}
write B (sorted by docIDs)
```

Figure 5: Value Based Clustering Algorithm of Layer-0

algorithm does not allow a value to span multiple posting lists. Furthermore, any posting list containing two or more distinct values will surely be smaller than F, respecting the $2F$ bound on filtering. Note that the algorithm does create blocks whose size is larger than F whenever a single value is shared by more than F documents. In such a case, a block will be dedicated to that value. Nevertheless, as large as they may become, these blocks never need to undergo filtering. Their value either falls within the query's range, requiring the entire block to be taken, or it does not, rendering the whole block as irrelevant to the query.

The above algorithm could fragment layer-0 to include some small posting lists (with rare values sandwiched between two popular values). However, since every two adjacent posting lists are at least of size $F + 1$, the overall number of layer-0 blocks is bounded by $2\lfloor\frac{N}{F+1}\rfloor + 1$.

5. EXPERIMENTS

This section shows expriments on the benefits and tradoffs of the L-layerer c-regular super block design. We have implemented this design in Trevi, an Intranet search enginge built in IBM [3]. Our implementation was done in C++ and our posting lists are implemented as read-only B-trees. Following the approach described in [6], we use a "packed" data representation, where the values for the same B-tree key are stored in fix length blocks in the leaf pages of the tree.

We ran all experiments in a Linux system with a 2.2GHz Intel Xeon Pentium 4 processor and 2 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 a real dataset form from the IBM intranet augmented with two parametric values per document that were syntetically generated. The first one was generated using a uniform distribution while the second one used a power law distribution with exponent two. The distributuions were derived indepenently and are not releated to the docIDs or to the order the documents are organized in the inverted index. We used sysntectic parametric fields to show that approach is indepenent of the value distrubitons.

We considered two index sizes, 100 thousand and 1 million documents. The index size for the IBM Internet is approaximatrly 5 million documents after duplicate elimination. However, parametric posting lists of size 100K and 1M documents are common, since not all documents have parametric fields. For each of these indices we built several configurations of the parametric postings, varying the number of extra layers and the clustering factor. For these experiments we fixed the filtering factor F by fixing the block sizes for layer 0 to 4K. In this configuration each block in layer 0 can hold approximatly 250 values, which caused the number of blocks in layer 0 to be 400 for the 100K document index and 4000 for the 1M documents index. We then varied L and c, computing the best possible c for each L using the result from proposition 1. The index configurations we used are summarized in Table 1. Besides the configurations listed in Table 1, we have also built indices no extra layers, for which the clustering factor is irrelevant.

	100K		1M	
L	*Theoretical c*	*Best c*	*Theoretical c*	*Best c*
2	5.74	4	12.56	16
3	3.76	4	6.68	8
4	2.88	4	4.57	4
5	2.33	2	3.54	4

Table 1: Index configurations tested

Index size and build performance. Table 2 shows the index index sizes for each index. It also shows the overhead of the parametric posting lists over the total index size. This overheads are heavily dependent on the number of parametric fields per document and grow linearly with this number. In this case we indexed only two parametric fields per document. Table 3 shows the index build performance for building the parametric posting lists. It also shows the time

overhead for building the parametric posting lists for the extra layers, which was in the worse case 12% for the 100K index and 9% for the 1M index. As expected, both the index size and the build time grow linearly with the number of layers. In addition, the impact of extra layers is not large, both in size and build time performance.

	100K		1M	
L	*Size (MB)*	*Overhead*	*Size (MB)*	*Overhead*
1	3.1	0.003	31	0.005
2	6.2	0.006	63	0.012
3	10	0.010	94	0.018
4	12	0.012	126	0.024
5	16	0.016	157	0.03

Table 2: Index sizes for the different index configrations

	100K		1M	
L	*Time (ms)*	*Overhead*	*Time (ms)*	*Overhead*
1	94377	0.118	400204	0.077
2	95183	0.119	414280	0.080
3	97070	0.122	428531	0.082
4	98623	0.124	443204	0.085
5	98575	0.124	456411	0.088

Table 3: Index build time for the different index configrations

Runtime performance. Figure 6 shows how the number of qualifying documents vary with the query range for the uniform and powerlaw parametric fields. Figures 7 to 10 show the query performance varying the number of extra layers for the two parametric fields in the 100K and 1M documents index. These graphs also show the runtime performance of using a naive filtering mechanism described in Section 2. The y-axis is ploted in log scale since the query performance for the filtering approach is orders of magnitude slower.

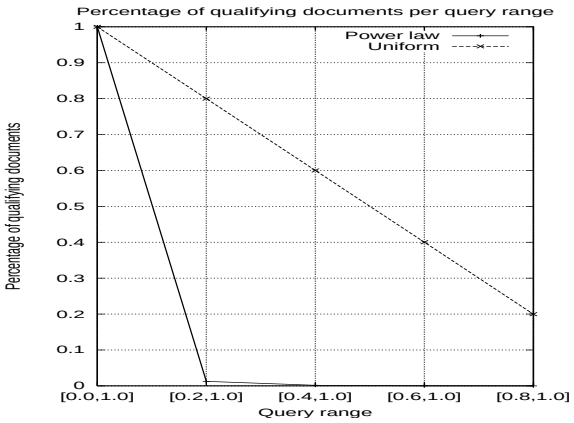

Figure 6: Result sizes varying the query selectivity for the two distributions

Figure 7 shows that using no extra layers ($L = 1$) is already about 100 times faster than the naive filtering approach and that using $L = 2$ is about two times faster than $L = 1$. However, increasing the number of layers to more than two does not improve performance for this index size. The reason is that for $L = 1$ the number of posting lists to merge is aready small. In this case the extra layers do not add sinificant benefit, although they also do not add noticible overhead. For the power law distritubtion, since the result sizes are smaller, $L = 1$ is already good enougth as shown in Figure 8.

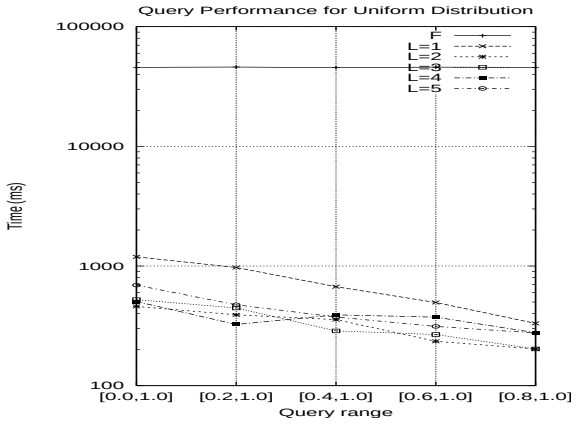

Figure 7: Query runtime performance for the different 100K index configurations for the uniform distribution

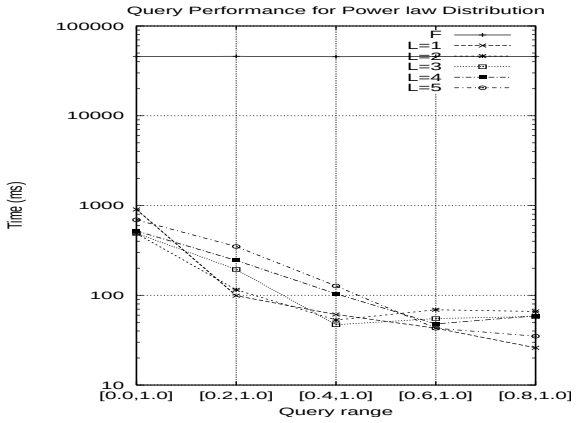

Figure 8: Query runtime performance for the different 100K index configurations for the power law distribution

For the 100K index the approach based on no replication ($L = 1$) is quite efficient. This is only true since the maximum number of blocks to be merged in that case is at most 400, which is not a very large number. For the 1M index, that is not the case. In fact, we for the uniform distribution and for query $[0.0, 1.0]$ in the power law distribution our implementation was not able to finish the queries using only the L-0 posting lists. Therefore we omit the results for $L = 1$ in Figures 9 and 10.

The results for the 1M index are similar to the ones for the 100K index. In this case the scheme with $L = 3$ presented the best results, while the schemes with extra layers did not show significant improvement or degradation in query pre-

formance. This shows that even for reasonably large parametric posting lists $L = 3$ is sufficient to produce oders of magnitude improvments over the naive solutions.

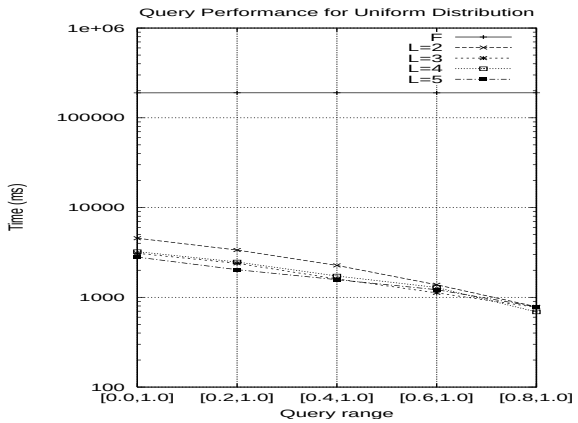

Figure 9: Query runtime performance for the different 1M index configurations for the uniform distribution

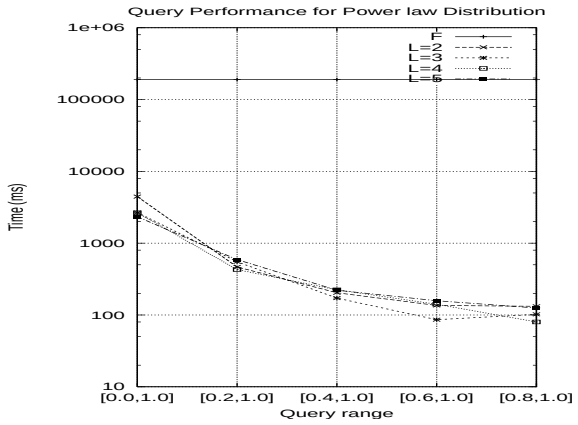

Figure 10: Query runtime performance for the different 1M index configurations for the power law distribution

6. VARIABLE CLUSTERINGS WITHIN LAYERS

The multi-layered scheme presented in Section 4.2 constructs an initial layer of posting lists (layer-0). For each subsequent layer $i (i \geq 1)$, it employs a clustering factor c_i, that defines how many posting lists of layer $i-1$ are merged into each list of layer i. Proposition 2 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 at times, clustering variable amounts of layer $i-1$ posting lists when constructing level i, can result in better performance. Specifically, it may lower the bound on the number of posting lists that need to be OR-ed when evaluating a query.

We begin with an example. Let $L = 1$ and $b = 50$. The optimal c-clustering design with 1 extra layer has $M_{min}(1, 50) = 18$, achieved when setting $c = 5$. However, clustering the 50 blocks of layer-0 in groups of $5, 6, 7, 7, 7, 7, 6, 5$ lowers M to 16. Figure 11 displays the two layouts.

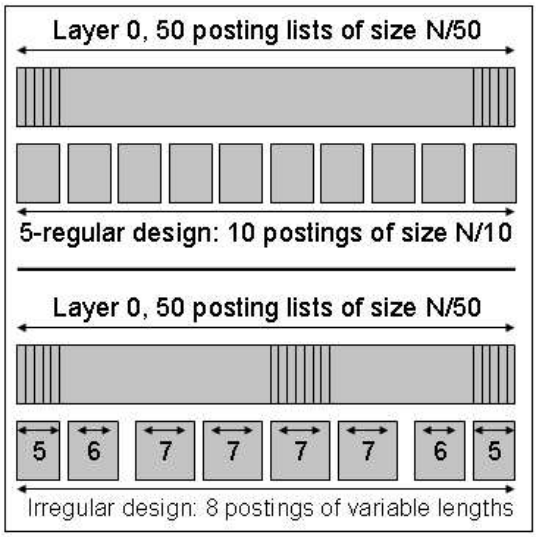

Figure 11: Regular vs. variable clustering factors

A full analysis of designs with variable clusterings within layers is beyond the scope of this paper and is left for future work. We do, however, outline the analysis when $L = 1$ for a given number of posting lists in layer 0, denoted by b_0. A design for layer 1 consists of a number of blocks b_1 and natural numbers $k_1, k_2, \ldots, k_{b_1}$, such that $\sum_{j=1}^{b_1} k_j = b_0$. The first k_1 blocks of layer 0 are merged to the first block of layer 1, the next k_2 layer-0 blocks form the second block of layer 1, and so forth.

For a given choice of b_1 and $\{k_1, \ldots, k_{b_1}\}$, let us evaluate the number of posting lists that we may need to access when a query defines a range that is covered by layer-0 blocks a_{min} through a_{max}. Denote by $\alpha_{min}[\alpha_{max}]$ the layer-1 block that incorporates $a_{min}[a_{max}]$. Evaluating the query will require the OR-ing of:

1. At most $k_{\alpha_{min}}$ layer-0 blocks that were merged into block α_{min} of layer 1.

2. At most $k_{\alpha_{max}}$ layer-0 blocks that were merged into block α_{max} of layer 1.

3. Layer-1 blocks $\alpha_{min} + 1$ through $alpha_{max} - 1$.

Thus, in order to minimize the worst case number of OR-ed postings for given values of L and b_0, one must optimize the following plan:

$$\min_{b_1, \{k_1, \ldots, k_{b_1}\}} \quad \max_{0 \leq i < j < b_1} \quad [k_i + k_j + (j - i - 1)]$$

$$\text{s.t.} \quad \sum_{j=1}^{b_1} k_j = b_0, \quad k_j > 0 \ .$$

7. CONCLUSIONS AND FUTURE WORK

TODO

8. REFERENCES

[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. 7th International WWW Conference*, pages 107–117, 1998.

[2] A. Broder, D. Carmel, M. Herscovichi, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Twelfth International Conference on Information and Knowledge Management (CIKM 2003), New Orleans, LA, USA*, pages 426–434, November 2003.

[3] M. Fontoura, J. Zien, E. Shekita, S. Rajagopalan, and A. Neumann. High performance index build algorithms for intranet search engines. In *VLDB 2004, Proceedings of 30th International Conference on Very Large Data Bases*. Morgan Kaufmann, 2004.

[4] L. Gravano, editor. *IEEE Data Engineering Bulletin, Special Issue on Text and Databases*, volume 24, 2001.

[5] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an rdbms for web data integration. In *Proceedings of the twelfth international conference on World Wide Web*, pages 90–101. ACM Press, 2003.

[6] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *Proceedings of the tenth international conference on World Wide Web*, pages 396–406. ACM Press, 2001.

[7] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C - The Art of Scientific Computing*. Cambridge University Press, 1988.

[8] S. Raghavan and H. Garcia-Molina. Integrating diverse information management systems: A brief survey. *IEEE Data Engineering Bulletin*, 24(4):44–52, 2001.

[9] S. Raghavan and H. Garcia-Molina. Complex queries over web repositories. In *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases*, pages 33–44. Morgan Kaufmann, 2003.