Bit-Vector Search Filtering with Application to a Kanji Dictionary

Matthew Skala

IT University of Copenhagen, Copenhagen, Denmark
mskala@ansuz.sooke.bc.ca

Abstract. Database query problems can be categorized by the expressiveness of their query languages, and data structure bounds are better for less expressive languages. Highly expressive languages, such as those permitting Boolean operations, lead to difficult query problems with poor bounds, and high dimensionality in geometric problems also causes their query languages to become expressive and inefficient. The IDSgrep *kanji* dictionary software approaches a highly expressive tree-matching query problem with a filtering technique set in 128-bit Hamming space. It can be a model for other highly expressive query languages. We suggest improvements to bit vector filtering of general applicability, and evaluate them in the context of IDSgrep.

1 Introduction

Many data structure problems of interest are specializations of the following general database query problem.

Problem 1. Given a universe U, preprocess a database $S \subseteq U$ into a data structure, to efficiently answer queries of the form "Find $Q \cap S$ for a given query Q." The database will be given by explicitly listing its n elements, but the query Q will be specified in some much more concise query language, which will not necessarily permit all arbitrarily-chosen subsets of U to be queries.

For example, in one kind of *similarity search* the query is described by a single element $q \in U$; the set Q to be intersected with S is then all elements "similar" to q, for some definition of similarity. In the present work, we are interested in database queries for which each element of S is or is not part of the query result independently—excluding such things as *nearest neighbour* queries, where the presence of one element in the database can affect whether some other element should or should not be returned.

The range of different query sets Q, determined by the expressive power of the query language, affects how this problem can be solved. A query language with very little expressive power, for instance consisting only of intervals of permitted values in a single numeric attribute, permits the use of tree-based data structures with typical $O(\log n)$ query time. An even less expressive language, such as one expressing only singleton sets (thus, membership queries) would be naturally solved using hash tables for constant query time. But when Q is specified using an *advanced* query language, loosely defined as one with great expressive power, it becomes more difficult to apply data structure techniques, and the data structures give less useful guarantees.

The naive solution to Problem 1 for an advanced query language would examine every element of S in $\Omega(n)$ time. Similarity queries of a linear-space data structure based on distance in a metric space also quickly approach $\Omega(n)$ number of elements examined when the dimensionality is large [5]; and because some problems arising from similarity search become $\mathcal{NP}$-hard in arbitrary dimension [21,26], and others have disappointingly large polynomial lower bounds associated with complexity-theoretic reductions from hard problems [28], there appears to be a connection between expressiveness of query languages and the dimensionality of the data. The language of high-dimensional near-neighbour queries is expressive enough to ask hard questions.

We are interested in query languages that embed Boolean operators: where the ability to specify query sets Q_1 and Q_2 implies also being able to specify $Q_1 \cap Q_2$ (AND), $Q_1 \cup Q_2$ (OR), and $U \backslash Q_1$ (NOT). Including such operators makes the language sufficiently expressive that it may be hard to beat the naive solution; and performance suffers even further if a complicated query language makes testing an element against the query an expensive operation in itself.

The present work describes and experimentally tests some techniques for speeding up general database query with expressive query languages, when bit-vector filtering is already in use. The hope is to reduce the constant in the $\Omega(n)$ by not examining *every* element in the database individually; and to possibly get better than $\Omega(n)$ performance on easy queries while still supporting hard queries on the same data structure. Our approach builds upon a bit-vector filtering method described previously [23] and implemented in a mature free-software project. We have a specific application originating in computational linguistics, but the techniques studied here are intended for general use.

1.1 About the Application

The Tsukurimashou Project [22] develops parametric fonts for Japanese-language typesetting, and associated software tools. IDSgrep is one of these tools: a search utility for Han character dictionaries [23].

The term *Han script* refers generically to a set of tens or hundreds of thousands of characters used in varying forms to write text in East Asian languages. Use of Han script is current in Chinese and Japanese. Korean is now written primarily in an alphabetic script with limited use of Han characters. Vietnamese is written in Latin script today, but it was historically written with Han characters, and specifically Vietnamese forms for Han characters are standardized in Unicode. The present work focuses on *kanji*, the Japanese form. One feature of Han script is that characters can be analysed as hierarchical combinations of elements that may be shared with other characters, or may be characters in themselves, as shown in Fig. 1.

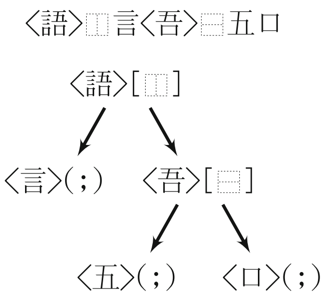

Fig. 1. A dictionary entry for the character 語 meaning "language" [23].

Font developers, language learners, and computational linguists each have reasons to query databases of these trees with Boolean criteria, as in "Find all characters that contain 言 on the left, and 口 anywhere, but not 上."

The details of the IDSgrep data model and query language are not new here and not relevant to the algorithmic considerations of the present paper, which treats the matching function as a black box. They are covered in earlier work [22,23] in much greater detail than is possible in this space or appropriate to the present work. But to summarize for interested readers: the structure of each character is represented by an *EIDS tree* ("Extended Ideographic Description Sequence," an extension of Unicode's IDS concept [27]), like the one illustrated in Fig. 1. Each node of the tree is labelled with a *functor* describing the relation among its children, such as ⿱ for one above the other, and optionally a *head* such as 吾 naming the character represented by that subtree. Not all subtrees of characters are characters in themselves, so not all nodes have heads. Queries are also trees in the same format, entered using a simple prefix syntax.

The basic matching rule is that if a query and a dictionary entry both have heads, then they match if and only if the heads are identical; if they do not both have heads, then they match if their functors and all their children match, recursively. But there are also special matching operators invoked by special functors in the query. For instance, ? is a match-everything wildcard, allowing the query ⿱?⺗ to match any dictionary entry for a character with a top and bottom part in which the bottom matches ⺗. Special operators include match-everything; Boolean AND, OR, and NOT; match any subtree; an *associative* match which rearranges trees along the lines of the associative law in arithmetic; and a few others intended for special purposes. Evaluating the full matching function between one query and one dictionary entry is a relatively expensive operation (worst-case cubic in the size of the tree and the query, excluding certain matching operators that invoke third-party software with no time guarantees); and a naive implementation of search would do this evaluation on all of the $O(n)$ trees in the database.

To support filtering search, IDSgrep calculates a *bit vector* for each EIDS tree, as shown in Fig. 2. Bit vectors, like EIDS trees, are treated as opaque by the search algorithm, but internally each one is the concatenation of four 32-bit

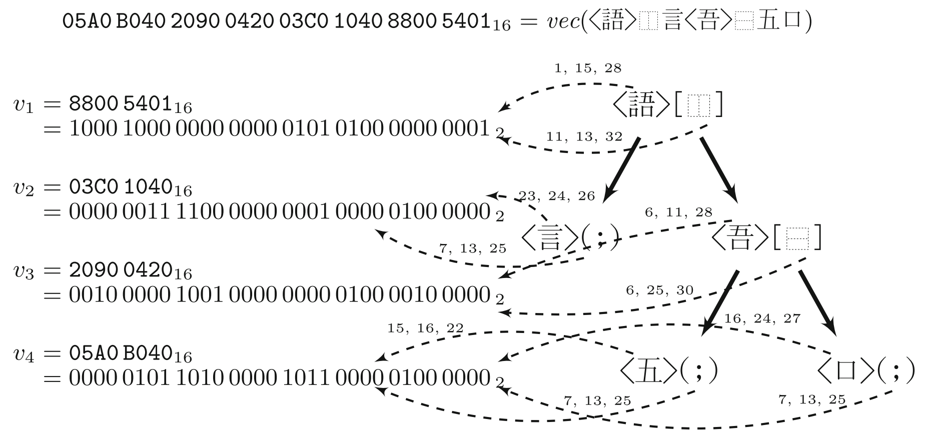

Fig. 2. Calculating a bit vector [23].

words which are Bloom filters of the tree's node labels. The first word, denoted v_1 in the figure, encodes the root of the tree. The functor ⿰ and arity (it is a binary node) are hashed to select bits 11, 19, and 32; the head 語 selects bits 1, 15, and 28. Similarly, the left child of the root determines v_2, the right determines v_3, and all other nodes select bits in v_4. Conditions on trees expressed in the query language, such as "must contain this label anywhere in the tree" imply conditions on the bit vectors, such as "at least three bits in this subset of the indices, must be set"; and the division into words allows computing the bit vector for a tree given its root labels and the vectors of the root's children.

1.2 Related Work

Bloom filtering [3] is well known. Guo et al. [9] describe the technique, also well known, of combining Bloom filters with Boolean AND and OR to perform the same operations on the sets the filters approximate. Our tree-matching problems are connected with *unification* in logic programming languages such as Prolog [4], and Aït-Kaci et al. [1] introduced bit vectors as a way to solve unification problems.

Advanced query languages for trees often take the form of modifications to the language of regular expressions. Polách describes such pattern matching in general abstract terms [18], and there is much work on regular expression-like tree matching specific to computational linguistics applications [7,14,15]. Some of our own work [24,25] applies bit vectors to unification in computational linguistics. Kaneta et al. [11] use them for another tree-matching problem. These references are described in more detail, with others related to *kanji* dictionaries and the linguistics application of IDSgrep, in our earlier IDSgrep paper [23].

The difficulty of high-dimensional queries has become known as the *curse of dimensionality*: exact query problems in high-dimensional spaces, across a wide range of different kinds of problems, have a strong tendency toward costs in time, space, or both that are exponential in the dimension. Approximation techniques have become standard in efforts to achieve practical results for high dimensions [10]. Theoretical work like that of Williams [28] links the difficulty of similarity search to the Strong Exponential Time Hypothesis, essentially saying that (for exact queries, in the worst case), we cannot do better than looking at $\Omega(n)$ points without proving unexpected deep results in complexity theory. Query languages that include Boolean logic have obvious direct application to satisfiability-type $\mathcal{NP}$-hard problems, and work like that of Frances and Litman [8] and our own [21,26] links similarity-based queries to $\mathcal{NP}$-hardness. Since advanced query languages seem doomed to hardness when the dimension is high enough, there is interest in at least *measuring* dimensionality of real data to detect when that phenomenon occurs [17,20].

The Binary Decision Diagram (BDD) is an interesting data structure in its own right, used here as a black box. Knuth describes its workings in detail [13]. IDSgrep's implementation uses the BuDDy library, due to Lind-Nielsen [16].

1.3 Notation

Although the software manipulates bit vectors as constant-sized objects using CPU bitwise instructions, we write them as if they were sets (implicitly, the sets of indices containing 1 bits) as a notational convenience. Thus, for bit vectors u and v, we write $u \cap v$ for the bitwise AND of u and v; $u \cup v$ for the bitwise OR; and $u \subseteq v$ for the statement that v contains a 1 bit at every index where u contains a 1 bit. We also write $|u|$ for the Hamming weight (or population count) of u, which is the number of 1 bits.

2 Bit-Vector Search with Enhancements

IDSgrep [23] performs a general database query (Problem 1) on opaque objects representing *kanji* dictionary entries and queries against them. Both happen to be EIDS trees as mentioned earlier, but the data structure is deliberately abstracted from the query algorithm. From the search algorithm's point of view, there is simply a database of arbitrary objects and a relatively expensive function $match(N, H)$ which returns a Boolean value true if and only if N (the *needle*, an EIDS tree representing the query) is considered to match H (the *haystack*, an object from the database). With only that abstract interface, nothing better than linear search would possible.

However, the underlying query language also provides bit vectors for the database objects, and filtering functions from the bit vectors to Boolean results, guaranteed to return true for all vectors of objects that match the query. A more efficient, but still linear, search can evaluate the filter functions first, and only invoke $match(N, H)$ when all filters return true. The bit vectors and filtering functions remain opaque, abstracted from the search algorithm.

The first layer of filtering uses a *lambda filter*, which is a pair (m, λ) consisting of a bit vector m and an integer λ, considered to match a dictionary entry's vector v if $|m \cap v| > \lambda$. Note that setting $\lambda = -1$ would match everything, giving a correct but unhelpful filter; the filter generator heuristically attempts to maximize λ and minimize m, while maintaining correctness.

The second layer of filtering uses a binary decision diagram directly encoding a monotonic function from bit vectors to Boolean truth values, again attempting to heuristically make that function return true as rarely as possible while still including all vectors of matching EIDS trees. This *BDD filter* is evaluated only if the lambda filter matches; then only if the BDD filter also matches does IDSgrep proceed to evaluating the exact EIDS matching function. As described in previous work, use of these filtering layers results in a significant improvement in query time on real-life data [23].

The new issue we address in the present work is that although the filtering can avoid many invocations of $match(N, H)$, the lambda filter at least is still tested for every entry in the database; and so there is a $\Omega(n)$ lower bound on all queries, though with an improved constant because of the filtering. How can the search algorithm on these opaque objects avoid looking at every entry?

2.1 Blocks with Bounds

Suppose the index file is divided into *blocks* and we record some information about each block summarizing logical statements that apply to all vectors in the block. If, based on that information, we can infer that *no* vector in a block could possibly match the query, then we can skip over examining the entire block. If the number of blocks is asymptotically smaller than the number of entries in the database, and we end up accepting (not skipping) less than a constant fraction of them, then the query time can break the $\Omega(n)$ barrier.

Bearing in mind the monotonic nature of the filtering layers, which implies that only 0 bits in a vector are really useful for excluding entries, we would like the summary to give maximal information about the 0 bits of the vectors in the block. We choose to store *containment* and *cardinality* bounds: for each block, a bit vector b and integer μ such that for every vector v in the block, $v \subseteq b$ and $|v| \leq \mu$. The optimal values are b equal to the OR of all vectors in the block and μ equal to the greatest Hamming weight. Note the similarity in structure between these bounds and the lambda filtering function.

Given bounds (b, μ) for a block and a lambda filter (m, λ), if $\min\{|b \cap m|, \mu\} \leq \lambda$ then it is impossible for any vector in the block to match the filter, and we can skip the entire block. Similarly, we can detect cases where an entire block fails to match a BDD filter, by a traversal of the BDD.

We report experimental results for a range of block sizes up to putting the entire database in a single block. Doing that is equivalent to not using blocks at all, except in the rare case of a match-nothing query that can be proven to match nothing from the whole-database containment and cardinality bounds.

2.2 Sorting

Since in our application the order of entries in the database is not significant, we can use that order to enhance search. In particular, it would cost very little to sort the index into lexicographically increasing order. If we are also splitting the index into blocks, we can sort within each block.

Given a lambda or BDD filter, we can easily compute the lexicographic range of vectors it could match, and then at query time, start with a binary search to find the first vector in the index or block that is within the range. Unfortunately, since both filters are monotonic, the all-ones vector will be matched by every filter that matches anything, and so only the lower end of the range usefully limits the search, and this improvement can only improve the constant in the search time. Nonetheless, it seems an inexpensive way to remove a few more filter checks from the search.

We compute the lexicographic bounds just once per query. In principle, a more detailed calculation could find lexicographic bounds on the vectors that match the query *and* that also obey the containment and cardinality bounds of a block, and this way we could usefully generate a lexicographic upper as well as a lower bound. But such a calculation would be more expensive in itself and would have to be repeated for every block instead of being done once per query. It seems unlikely to give much benefit in practice over the other ways we are already applying the containment and cardinality bounds. We leave testing that for future work.

Sorting the index data by bit vector implies that our eventual access to the dictionary file will be in randomized order, and random seeking could significantly increase the overall cost of the search when the dictionary file is stored on disk. To avoid this issue, when sorting the index we also generate a sorted dictionary with the entries arranged in the same order as the index; then accesses to dictionary entries during search will at least be sequential, if not consecutive.

2.3 Clustering

It ought to be the case that vectors in the same block are similar to each other, for some definition of similarity. Then queries are more likely to match either none or many of the vectors in the block, maximizing the chance that we can exclude the block on the basis of its summary information. If we must keep vectors in their original database order then we are stuck with making blocks be consecutive intervals of that order; but if we are sorting the index anyway, it makes sense to do clustering first and make the blocks correspond to clusters. Even the sorting itself, followed by making blocks out of consecutive entries, might be expected to provide some clustering benefit because similar vectors might tend to appear near each other in lexicographic order.

Noting the way the containment bound works as a bitwise OR of the vectors in the block, it seems the worst situation is when we include a vector in a large block that has a 1 at an index where all others in the block are 0. Then just because of including that vector, we have an additional 1 in the containment bound and fewer chances to skip all the other vectors in the block. We tested a variant of k-means clustering using a special similarity measurement that captures the idea of avoiding such situations.

Suppose, during clustering, we are considering moving a vector v into a cluster C_i, which is a multiset of vectors. If v is already assigned to C_i, let C'_i be the cluster with (one instance of) v removed; otherwise $C'_i = C_i$. Then we compute:

$$\mathit{fit}(v, C'_i) = \frac{1}{|C'_i| + 1} \left[\min_{v_j = 1} |\{w \in C'_i | w_j = 1\}| + \begin{cases} 1 & \text{if } v \text{ was assigned to } C_i, \\ 0 & \text{otherwise.} \end{cases} \right]$$

This says that how well a vector fits in a cluster is basically the fraction of other vectors in the cluster that share its least popular attribute; a vector will fit well in a cluster where all its 1 bits are already included in the cluster's bound as a result of many other vectors. The special handling for the case of v "already" assigned to C_i appears to be necessary for reliable termination; other variations we tried would loop indefinitely on some inputs. For similar reasons, we limited cluster size to at most twice the initial block size. Any cluster currently at the maximum size limit will not be considered as a possible destination for moving a vector during the optimization. That counteracted an observed tendency for the algorithm to put most of the database in a single huge cluster.

The clustering algorithm starts by assigning consecutive blocks of vectors from the index to be clusters (in input order if unsorted, or sorted order if we are sorting) and then iteratively attempts to move every vector to the cluster that maximizes its value of $\mathit{fit}(v, C'_i)$, until no more such moves are possible. If using sorted indices, we sort within the clusters again after finding them.

3 Experimental Results

We extended the current version of IDSgrep to use the techniques described in the previous section, creating a special version for these experiments designated version 0.5.2, and we evaluated it using the same data, test queries, and hardware and operating system configurations used in our earlier work [23]. The test database contains 217,288 entries for decompositions of Han characters especially concentrating on Japanese *kanji*, gathered from the CJKVI [12], CHISE-IDS [6], KanjiVG [2], and Tsukurimashou [19] projects. There are 1,642 test queries, representing a spectrum of complexity and result set size from single exact-match character queries to more complicated Boolean operations. Speed testing was performed on a MacBook Pro equipped with a 2.3 GHz Intel iCore i7 CPU and 8 G of RAM, running Mac OS X 10.9.5. A package of our code and data is available to assist in reproducing the results, and includes the experimental results that were omitted here for space reasons.

The test query set from our earlier work [23] was designed to demonstrate the dictionary application, and includes many queries for which lambda and BDD filtering are ineffective. Those queries also tend to be relatively slow in other parts of the software; as a result, they dominate the total real running time for the test set taken as a whole when filtering is applied, and they make time measurements specific to filtering difficult. To better measure the filtering-specific techniques in the present paper, we separated out 512 "slow" queries, which are those with associative-match or match-anywhere operators at the root, or consisting solely of Boolean combinations of such queries. Such queries are easily recognized by testing that definition, and their important feature is that they generate filter functions which match nearly everything, so filtering search has little effect. The remaining 1,130 queries, where filtering search is expected to be of more use, are designated "fast." We tested each power of two block size from 4 to 262,144; that last, being larger than the database, effectively means no blocking at all.

Running the 1,130 queries in the fast query test set on the database of 217,288 entries means doing, or avoiding, a total of $1{,}130 \times 217{,}288 = 245{,}535{,}440$ tests of whether a query matches a database entry. Each test either results in the entry being discarded at some level of processing, or in a final match which returns a result. Figure 3 shows how frequently these outcomes occurred in our experiments, for selected choices of parameters. The categories shown in each stacked column represent increasingly expensive outcomes. A match may be discarded by skipping a block; by binary search (only for sorted indices); by the lambda filter; by the BDD filter; or it may be a BDD hit, which is true of 894,341 tests (0.36 %) for all parameter settings. Of those BDD hits, 29,606 will match in the final tree test, but that is too small a proportion (0.012 %) to usefully depict on the chart. Results shown in this figure are the same for all trials of each parameter set because the query algorithm is deterministic. A similar chart for the 512 slow queries is shown in Fig. 4. Here also, the final tree matches (137,415, or 0.12 % of $512 \times 217{,}288 = 111{,}251{,}456$ tests) form too small a proportion to be visible on the chart.

The running times for the fast and slow query test sets are shown in Figs. 5 and 6 respectively. Note these figures use logarithmic scales. The quantities plotted are sample means of user CPU seconds per loop as measured by IDSgrep's built-in statistics feature, with error bars representing intervals of ± 2 sample standard deviations, on 20 trials of each combination of parameter settings.

4 Discussion and Conclusions

The outcome counts of Fig. 3 show that the new approach of excluding blocks based on their containment and cardinality bounds allows the query to exclude a significant fraction of more expensive filter checks, for the queries in the "fast" set where filtering is effective. Without sorting or clustering, this effect is only significant at the smallest block sizes, but with sorting, clustering, or both, we can exclude many blocks even at block sizes up to thousands of vectors. For sorted indices, the lexicographic lower bound allows excluding a few more vectors, up to about 10 % on the largest block sizes.

However, these techniques have very little effect on the "slow" queries shown in Fig. 4. We are only excluding blocks at all at the smallest block sizes, we

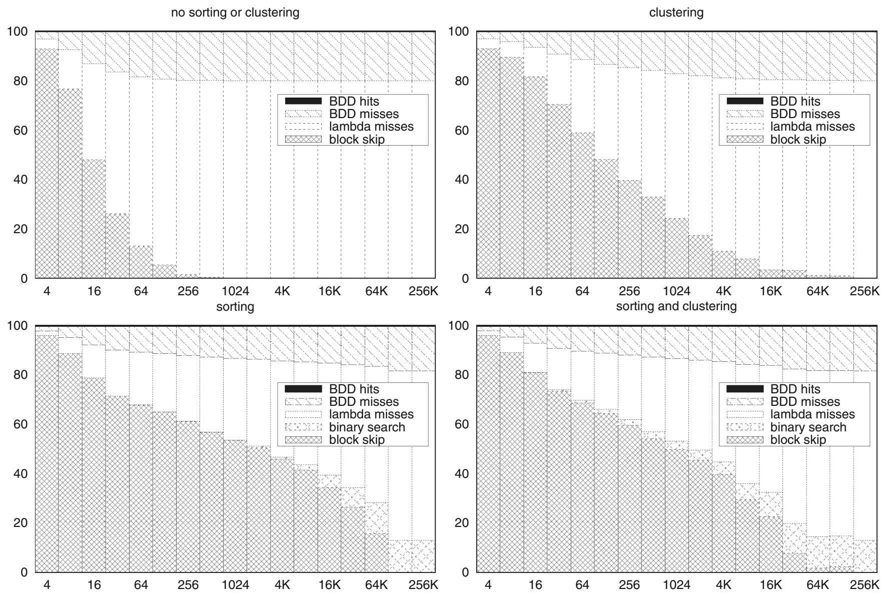

Fig. 3. Outcomes of testing dictionary entries against fast queries. Vertical axis: proportion in percent; horizontal: entries per block, 1K = 1,024.

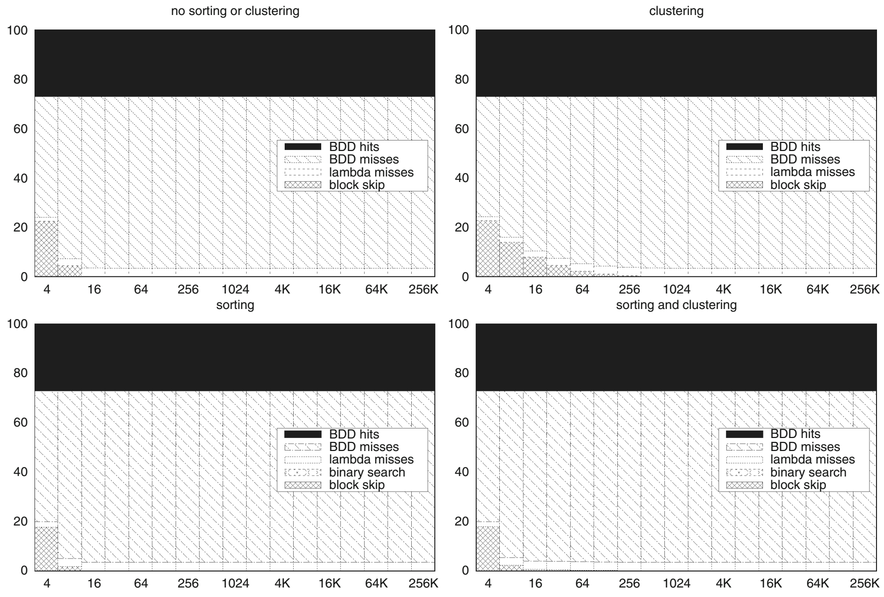

Fig. 4. Outcomes of testing dictionary entries against slow queries. Vertical axis: proportion in percent; horizontal: entries per block, 1K = 1,024.

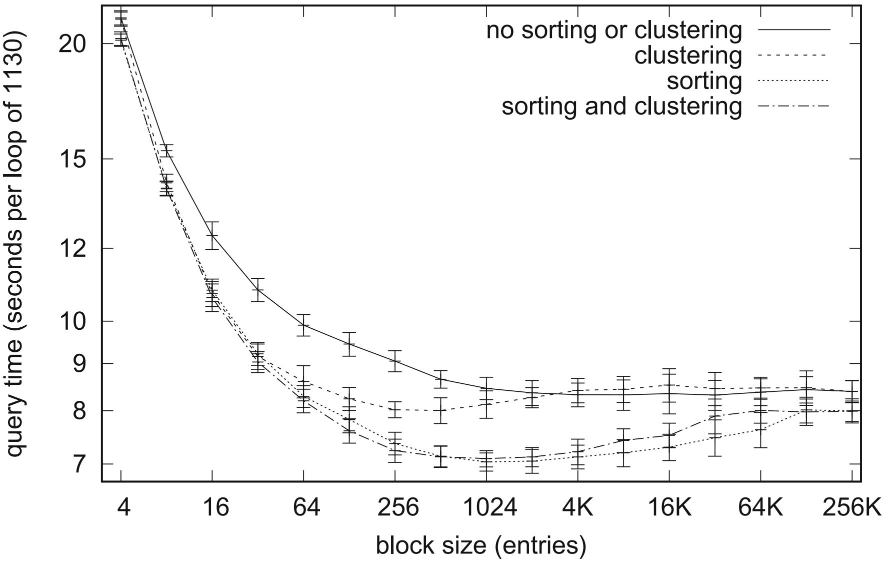

Fig. 5. Running times for fast queries.

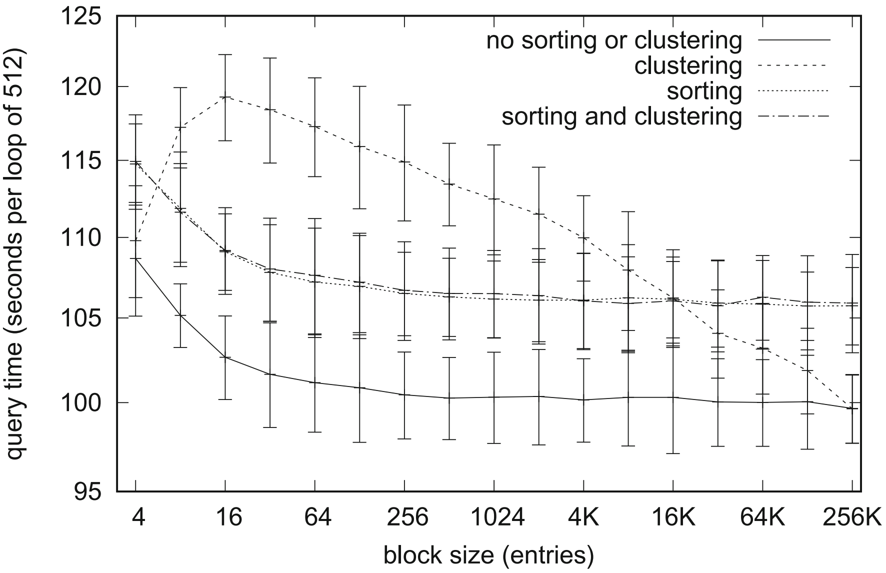

Fig. 6. Running times for slow queries.

are excluding very few blocks, and somewhat oddly, the clusters found without sorting first perform noticeably better than any other choice of options. These effects highlight the different nature of the “slow” queries: they are queries for which almost all vectors are lambda filter hits, and about a quarter of vectors are also BDD filter hits. Trimming the time consumption of filter *misses* makes very little difference to the bottom line performance.

To some extent we can say that IDSgrep is a victim of its own success. The plain $\Omega(n)$ search algorithm with lambda and BDD filtering is already so good, and in particular has such small constants for the filtering step, that it already shifts much of the running time away from filtering and into the final tree tests and ancillary tasks like parsing the file formats. For that reason, we should not expect to see the trends shown in Figs. 3 and 4 to be reflected strongly in the overall times of Figs. 5 and 6. The outcome counts measure only filtering, whereas the overall times also include parsing and tree matches, which do not vary between the experimental treatments.

The results suggest two directions for future work. First, more advanced variations of bit vector query are best targeted to larger data sets, and applications beyond *kanji* dictionaries; future work on bit vector queries in general might better use other applications for testing. Skipping blocks, even if it saves little time for IDSgrep, can reasonably be expected to help more when failing to skip blocks is more expensive, as in a very large database or one with an even more expensive query language than ours. It would be interesting to apply these techniques to advanced query languages on other, much larger, data collections—for instance, document databases with bits encoding keyword presence and Boolean queries over those, or image local-descriptor databases with very expensive similarity measurements. The smaller the result set in comparison to the overall database, the more we can realistically expect bit vector filtering to help.

Second, it may be appropriate to change the underlying bit vector and filter definitions in the specific application of *kanji* dictionaries. We are getting good filtering on some queries, but not on others, with 128-bit vectors; and the specific design and parameters of how EIDS trees generate bit vectors have not been changed or studied systematically since IDSgrep first introduced bit vector filtering. It would not cost much more time or space to switch to 256-bit vectors; could that bring more of the “slow” queries into the “fast” group? Can the definition of “slow” queries inform future bit vector designs that could exclude more entries and benefit more from the present work?

The difficulty appears to come from the associative and match-anywhere operators, which notably are not Boolean operators of the kind most likely to give hardness reductions. Maybe a redesigned bit vector definition specifically targeting those operators could move more queries into the “fast” class. In particular, the associative match operator currently generates a match-everything filter, but might be enhanced to generate a more restrictive filter. The match-anywhere operator in a query reduces to a Boolean OR of several other queries, naturally hitting many vectors, and an improved vector design might add some bits specifically to serve this operator better.

Requiring bit vector filters to be monotonic was an important technical aspect of the IDSgrep design. It made the difference between the feasibility and infeasibility of using BDDs. However, it limits the benefit of our new sorting technique, because monotonicity means we can only compute a useful lexicographic *lower* bound. The upper bound is always the all-ones vector. If we could make non-monotonic filters work with the rest of the system, it might help toward the goal of better than linear query time for easy queries. In particular, if a query matching only one entry could also match only one vector then we could expect the binary search to find that one vector after only $O(\log n)$ steps.

It is natural to extend the single-level blocking done here to a recursive tree-like structure, with a constant number (rather than constant size) of blocks each divided into smaller blocks, through as many levels as needed. A recursive data structure would lend itself to proving sublinear bounds for queries where that may be possible, while still degrading gracefully to linear time on harder queries where the lower bounds forbid anything better.

To conclude, we have proposed new techniques for filtering bit vector search, and tested them in the specific application of a *kanji* dictionary. The new techniques are experimentally shown to allow skipping a significant fraction of more expensive vector tests on some kinds of queries, but they cost enough to be of limited use for small databases or for queries where bit vectors are already failing. We have described technical issues relevant to our implementation. We have discussed these results and their implications both for bit vector query in general and the *kanji* dictionary application in particular.

References

1. Aït-Kaci, H., Boyer, R.S., Lincoln, P., Nasr, R.: Efficient implementation of lattice operations. ACM Trans. Program. Lang. Syst. **11**(1), 115–146 (1989)
2. Apel, U.: KanjiVG. http://kanjivg.tagaini.net/
3. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
4. Bramer, M.: Logic Programming with Prolog, 2nd edn. Springer, London (2013)
5. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. **33**(3), 273–321 (2001)
6. CHISE project. http://www.chise.org/
7. Choi, Y.S.: Tree pattern expression for extracting information from syntactically parsed text corpora. Data Min. Knowl. Disc. **22**(1–2), 211–231 (2011)
8. Frances, M., Litman, A.: On covering problems of codes. Theor. Comput. Syst. **30**(2), 113–119 (1997)
9. Guo, D., Wu, J., Chen, H., Yuan, Y., Luo, X.: The dynamic bloom filters. IEEE Trans. Knowl. Data Eng. **22**(1), 120–133 (2010)
10. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing. pp. 604–613. ACM, New York (1998)
11. Kaneta, Y., Arimura, H., Raman, R.: Faster bit-parallel algorithms for unordered pseudo-tree matching and tree homeomorphism. J. Discrete Algorithms **14**, 119–135 (2012)

12. Kawabata, T.: IDS data for CJK unified Ideographs. https://github.com/cjkvi/cjkvi-ids
13. Knuth, D.E.: The Art of Computer Programming, Pre-fascicle 1B, vol. 4. Addison-Wesley, Reading (2009)
14. Lai, C., Bird, S.: Querying linguistic trees. J. Logic Lang. Inf. **19**(1), 53–73 (2010)
15. Levy, R., Andrew, G.: Tregex and Tsurgeon: tools for querying and manipulating tree data structures. In: Calzolari, N., Choukri, K., Gangemi, A., Maegaard, B., Mariani, J., Odijk, J., Tapias, D. (eds.) 5th International Conference on Language Resources and Evaluation (LREC 2006), Genoa, Italy, 22–28 May 2006
16. Lind-Nielsen, J.: BuDDy: a BDD package. http://buddy.sourceforge.net/manual/main.html
17. Ott, E.: Chaos in Dynamical Systems, 2nd edn. Cambridge University Press, Cambridge (2002)
18. Polách, R.: Tree pattern matching and tree expressions. Master's thesis, Czech Technical University in Prague (2011)
19. Skala, M.: Tsukurimashou font family and IDSgrep. http://tsukurimashou.osdn.jp/
20. Skala, M.: Measuring the difficulty of distance-based indexing. In: Consens, M., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 103–114. Springer, Heidelberg (2005). doi:10.1007/11575832_12
21. Skala, M.: On the complexity of reverse similarity search. In: Chávez, E., Navarro, G. (eds.) First International Workshop on Similarity Search and Applications (SISAP 2008), Cancun, Mexico, 11–12 April 2008, pp. 149–156. IEEE (2008)
22. Skala, M.: Tsukurimashou: a Japanese-language font meta-family. TUGboat **34**(3), 269–278. In: Proceedings of the 34th Annual Meeting of the TEX Users Group (TUG 2013), Tokyo, Japan, 23–26 October 2013 (2014)
23. Skala, M.: A structural query system for Han characters. Int. J. Asian Lang. Process. **23**(2), 127–159 (2015)
24. Skala, M., Krakovna, V., Kramár, J., Penn, G.: A generalized-zero-preserving method for compact encoding of concept lattices. In: 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010), Uppsala, Sweden, 11–16 July 2010, pp. 1512–1521. Association for Computational Linguistics (2010). http://www.aclweb.org/anthology/P10-1153
25. Skala, M., Penn, G.: Approximate bit vectors for fast unification. In: Kanazawa, M., Kornai, A., Kracht, M., Seki, H. (eds.) MOL 2011. LNCS (LNAI), vol. 6878, pp. 158–173. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23211-4_10
26. Skala, M.A.: Aspects of metric spaces in computation. Ph.D. thesis, University of Waterloo (2008)
27. Unicode Consortium: Ideographic description characters. In: The Unicode Standard, Version 6.0.0, Section 12.2. The Unicode Consortium, Mountain View, USA (2011). http://www.unicode.org/versions/Unicode6.0.0/ch12.pdf
28. Williams, R.: A new algorithm for optimal 2-constraint satisfaction and its implications. Theor. Comput. Sci. **348**(2–3), 357–365 (2005)