Index Structures for Preference Database Queries

Markus Endres(✉) and Felix Weichmann

Institute for Computer Science,
University of Augsburg, Universitätsstr. 6a, 86159 Augsburg, Germany
endres@informatik.uni-augsburg.de,
felix.weichmann@student.uni-augsburg.de
http://www.dbis.informatik.uni-augsburg.de

Abstract. Preference queries enable satisfying search results by delivering best matches, even when no tuple in a dataset fulfills all preferences perfectly. Several methods were developed for preference query processing, such as window-based, distributed, divide-and-conquer, and index-based algorithms. In particular, all index-based algorithms were designed to evaluate Pareto preferences, where the participating preferences are all equally important. In this paper we present index structures for base preferences. Our comprehensive experiments show how indexing data for preference database queries enable faster access of the data tuples and therefore lead to performance advantages when evaluating preferences.

1 Introduction

Preferences are a well established framework to create personalized information systems. Skyline queries [3] are the most prominent representatives of preference queries. An implementation of preferences in database systems is PreferenceSQL [7] and the commercial product EXASolution [9] as well as a prototype of the Microsoft SQL Server [4]. Preferences in database systems are modeled as strict partial orders and a preference query returns the maximal elements according to this order, i.e., those tuples from the dataset which are not dominated w.r.t. the given preference.

Example 1. Assume the sample dataset in Table 1 and the wish for a car with *highest power* and a *price between 34000 and 37000 USD*, where both preferences should be considered as equally important (a Pareto preference). Then this query would identify the tuples with ID 4 and 5 as best objects. The tuple with ID 4 has a perfect match concerning the price, but the power of the tuple with ID 5 is higher. Therefore, both tuples are indifferent and form the result set.

Search efficiency is the most important performance criteria to preference query processing. In addition, as preference queries have been considered as an analytical tool in some commercial database systems [4,9], and the datasets to be processed in real-world applications are of considerable size, there is definitely the need for improved query performance. Indexing data is one natural

© Springer International Publishing AG 2017
H. Christiansen et al. (Eds.): FQAS 2017, LNAI 10333, pp. 137–149, 2017.
DOI: 10.1007/978-3-319-59692-1_12

Table 1. Sample dataset of cars.

Car	id	Make	Color	Power	Price
	1	BMW	Green	180	35000
	2	Audi	Green	170	32000
	3	Mercedes	Blue	200	38000
	4	BMW	Blue	230	34000
	5	Mercedes	Black	250	20000
	6	Mitsubishi	Black	120	50000
	7	Mitsubishi	Black	140	53000
	8	Audi	Eed	150	37000

choice to achieve this performance improvement. The advantage of index-based algorithms is that they need to access only a portion of the dataset to compute the Skyline, while non-index-based algorithms have to visit the whole dataset at least once. However, index-based algorithms have to incur additional time and space costs for building and maintaining the indexes. In addition, unlike most existing algorithms that require at least one pass over the dataset to return the first interesting point, indexing data for preference queries can be used to progressively return interesting points as they are identified.

For the evaluation of Pareto preference queries as in Example 1 there exist several index structures, see [5] for an overview. However, all these index structures were exclusively designed for Skyline/Pareto queries. In this paper we refer to *indexing techniques* for *base preferences*. We do not consider indexes for Pareto preferences as in previous work, but offer indexing methods based on common index structures for simple preference database queries and show how they perform against state-of-the-art preference computation algorithms. To the best of our knowledge there exist no other index structures for base preferences as described in this paper. Hence, this is the first work on this topic and our comprehensive experiments show the performance advantage in several synthetic and real world use cases.

The rest of the paper is organized as follows: Sect. 2 introduces preferences in database systems and presents common index data structures for database queries. In Sect. 3 we discuss the applicability of the common index structures to preference queries. Section 4 reports our comprehensive experiments, and Sect. 5 concludes with a summary and outlook.

2 Background

2.1 Preferences in Database Systems

Following [6], a *preference* P a *strict partial order* on the attribute list A. The result of a preference is computed by the *preference selection* and is called *Best-Matching-Only* (BMO) set. The BMO-set contains all tuples t from an input relation R which are not dominated w.r.t. the preference P: To specify a database preference, a variety of intuitive constructors have been defined. Preferences on single attributes are called *base preferences.* There are *base preference constructors* for *discrete* (*categorical*) and for *continuous* (*numerical*) domains.

Numerical Preferences. The interval preference $\text{BETWEEN}_d(A, [low, up])$ expresses the wish for a value between a *lower* and an *upper* bound. If this is infeasible, values having the smallest distance to $[low, up]$ are preferred, where the distance is discretized by the discretization parameter d. In the case of $d = 0$ the *distance* describes how far the domain value v is away from the optimal value. A d-parameter $d > 0$ represents a discretization of the distance to v, which is used to group ranges of attribute values together. Choosing $d > 0$ effects that attribute values inside d-intervals "left and right" of $[low, up]$ become indifferent, c.f. [6]. Specifying $low = up$ $(=: z)$ in BETWEEN_d we get the $\text{AROUND}_d(A, z)$ preference, where the desired value should be z. Furthermore, the constructors $\text{LOWEST}_d(A, \inf_A)$ and $\text{HIGHEST}_d(A, \sup_A)$ prefer the minimal and maximal values within the distance d, where $\inf_A$ and $\sup_A$ are the infimum and supremum of the attribute values. In the preference $\text{ATLEAST}_d(A, z)$ the desired values should be greater or equal to z. If this is infeasible, values within a distance of d are acceptable. $\text{ATMOST}_d(A, z)$ is its dual preference.

Categorical Preferences. A $\text{LAYERED}_m(A, (L_1, \ldots, L_m))$ preference on a categorical domain expresses that a user has a set of preferred values given by the disjoint sets L_i. Thereby the values in L_1 are the most preferred values, L_2 are the second choice, and so on. There are several sub-constructors of LAYERED_m. The positive preference $\text{POS}(A, \textit{POS-set})$ for example is defined as $\text{LAYERED}_2(A, (\textit{POS-set},\ \text{dom}(A) \setminus \textit{POS-set}))$ and expresses that a user has a set of preferred values given by the *POS-set.* The negative preference $\text{NEG}(A, \textit{NEG-set})$ is the counterpart to the POS preference. It is possible to combine these preferences to POS/POS or POS/NEG.

Example 2. Consider Table 1. If we specify $P_1 := \text{AROUND}_5(\text{power}, 130)$, the result are the cars with ID 6 and 7, because both have the same distance to the desired value and there does not exists a perfect match. The wish for an Audi or BMW leads to $P_2 := \text{POS}(\text{make}, \{\text{Audi}, \text{BMW}\})$. The result is $\text{ID} \in \{1, 2, 4, 8\}$.

It is possible to combine several base preferences into more complex preferences, where one has to decide the relative importance of the given preferences. Equal importance is modeled by the *Pareto preference*, whereas for ordered importance we use *Prioritization.* Both are not topic of this paper, but are discussed elsewhere.

2.2 Index Data Structures

In this section we recapitulate well-known index structures for databases which also can be applied to preference queries. Note that there are many other indexes, but the mentioned structures are practical for preference queries as well.

Range Trees have been designed to answer range queries efficiently [8]. They are similar to a B$^+$-trees, where all values in the left child are less than or equal to the value maintained at the node, while all values in the right child are greater. In a range tree all the data is stored in the leaves. Thereby, (1) the leaves of the tree are maintained in a *sorted order*, and (2) the leaves are *linked* to the next and previous nodes. The combined effect is that the data points form a sorted doubly linked list. A range tree is a balanced search tree and hence the search time is in $\mathcal{O}(\log(k) + |S|)$, k the number of values in the tree and $|S|$ the number of tuples in the answer set.

Hash Index uses a *hash function* $h(v)$ that takes a search key v and computes an integer in the range 0 to $B-1$, where B is the number of *buckets* [2]. A *bucket directory* holds the headers of B linked lists, one for each bucket of the array. If a tuple has search key v, then we store the tuple by linking it to the bucket list for the bucket numbered $h(v)$. Hash functions complete searching for *any* key in $\mathcal{O}(1)$ time, since one only has to lookup $h(v)$ for a search key v. Thus, in a database querying context, hash functions are desirable for *exact match queries*, but cannot support range queries well.

Trie Index (from re*trie*val) is used to index strings and to support efficient evaluation of categorical preferences. The root of a trie [2] (also known as Prefix B-tree [1]) represents the empty string. Each edge defines the next character of a string. The last character ends in a leaf node. Hence, every path from a root to a leaf encodes a string. The complexity of a search in a trie is given as $\mathcal{O}(l)$, where l is the length of the search string.

3 Indexes for Preference Queries

When considering indexing for *base preferences*, we have to keep in mind, that until now the fastest method to evaluate such preferences is a *linear scan* over the dataset and always store the temporarily best tuples in a set S. At the end of the scan S contains the BMO objects. The runtime complexity is $\mathcal{O}(n)$, where n is the size of the dataset. This linear scan is a modified version of the well-known BNL algorithm [3], which was developed for Skylines. Keep in mind that the costs for building and maintaining the index structures for preferences are the same as in the original data structures.

3.1 An Index for Numerical Preferences

Since all numerical preferences are sub-constructors of BETWEEN$_d$, it is enough to present an index structure for this preference and to discuss the search in the special cases for all other preferences.

If the query is a range query like BETWEEN$_d(A, [low, up])$, index structures like binary search trees, quadtrees, K-d-trees, and Hash index fail. For example, for the range query in a binary search tree, at a node, both branches may need to be traversed. If the query is for all points greater than a value, then the search degenerates to traversing all the branches and nodes of the tree, thereby suffering a performance worse than that of linear scan. If we use hashing, then we get no help for queries with large ranges. For example, if attribute A is restricted to range $a \leq A \leq b$, then we must look in the buckets for every value between a and b for possible values of A. There may easily be more values in this range than there are buckets, meaning that we must look in all, or almost all, the buckets. Hence, we need another data structure which is feasible for range queries.

Extended Range Trees. For numerical preferences we use a modified Range tree. Our extended Range tree (cp. Fig. 1) consists of a B^+-tree, where the leaves build a doubly linked list in a sorted order. Additional references to the first and the last element of the doubly linked list complete our modification.

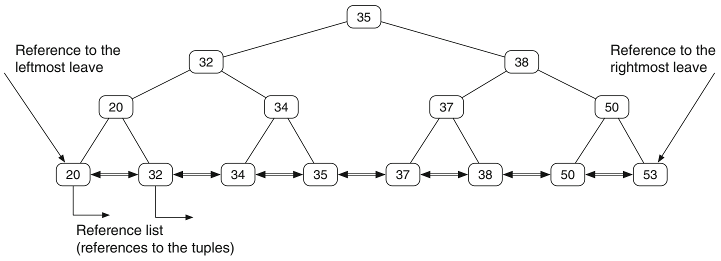

35
32
38
20
34
37
50
20
32
34
35
37
38
50
53
Reference to the leftmost leave
Reference to the rightmost leave
Reference list (references to the tuples)

Fig. 1. A one-dimensional Range tree for the price (in thousands) in Table 1.

Thus an evaluation for a BETWEEN$_d$ preference consists of a lookup in the tree, iteration over part of the doubly linked list and the return of the reference list. Additionally the index holds references for the first and last element of the doubly linked list such that a lookup in the index can be skipped for the LOWEST$_d$, HIGHEST$_d$, ATMOST$_d$, and ATLEAST$_d$ preferences.

Index-Based Evaluation of Numerical Preference Queries. We now have to discuss how the search does work in detail. The correctness of the search procedure is guaranteed by the fact that the leaves are sorted.

BETWEEN$_\mathbf{d}$(A, [low; up]): If we want to answer BETWEEN$_d(A, [low; up])$ preference queries, we search for all points that are $\geq low$ and $\leq up$. The query proceeds by first searching for the leaf that has the largest value just less than or equal to low. It then traverses all the leaves using the forward pointers until a leaf that is just greater than up is reached. If no tuple in $[low; up]$ is found, i.e., there is no match in the dataset, two cases may occur:

- $\mathbf{d = 0}$: $d = 0$ means that the tuples with the lowest distance to the interval $[low; up]$ are the preferred values. Consider the direct leaves left l and right r of $[low; up]$ and compute their distance $d_l(low, l) = low - l$ and $d_r(up, r) = r - up$. The index with the shortest distance corresponds to the preferred values.

– $\mathbf{d > 0}$: the preferred values lie in $[low - c \cdot d; low] \cup [up; up + c \cdot d]$, $c = 1, \ldots,$ until $c \cdot d$ reaches the infimum/supremum of the domain of A. The intervals with the lowest c contain the best matches.

$\mathbf{AROUND_d(A, z)}$**:** In AROUND$_d(A, z)$ the desired value should be z. If this is infeasible, values within a distance of d are acceptable. Hence, we first search for an exact match of z in the Range tree and if nothing is found, we continue as with the BETWEEN$_d$ preference.

$\mathbf{ATLEAST_d(A, z)}$ **and** $\mathbf{ATMOST_d(A, z)}$**:** These preferences correspond to a search in $[z; +\infty]$ and $[-\infty; z]$, respectively. For the ATLEAST$_d$ preference the search proceeds by returning all leaves that can be traversed using the backward pointers from the rightmost leave until an index entry $\leq z$ is found. For this we use the additional references to the doubly linked list. If z is not a node in the tree structure, the ATLEAST$_d$ preference returns the values with the lowest distance to z. Hence, we can proceed as above but have first to check if the rightmost leave is less than z. ATMOST$_d$ can be evaluated analogously. Note that without the additional pointers to the leftmost and rightmost leaves in the doubly linked list the search can be done similar to BETWEEN$_d$.

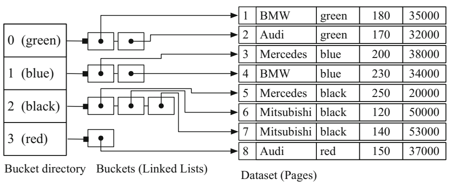
$\mathbf{LOWEST_d(A, inf_A)}$ **and** $\mathbf{HIGHEST_d(A, sup_A)}$**:** If $d = 0$ in LOWEST$_d$ we just search for the minimum in the dataset, i.e., find the lowest value in the Range tree. This can be done by the additional references to the doubly linked list. For $d > 0$ we proceed as in AROUND$_d$. Analogously with HIGHEST$_d$.

3.2 Indexes for Categorical Preferences

For the evaluation of categorical preferences we can use index structures which support *exact match queries*, because we always search for a perfect match in the index. All categorical preferences like POS, POS/NEG, etc., can be modeled using the LAYERED$_m(A, (L_1, \ldots, L_m))$ constructor. Hence, it is sufficient to consider indexing techniques for this preference.

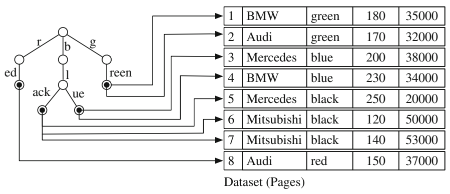
Hash Index. If we want to evaluate a LAYERED$_m(A, (L_1, \ldots, L_m))$ preference, the idea is to search for all objects $t \in L_i$ successively, i.e., we do an exact match query for each t until we find a match. This can be seen in the following example.

Example 3. Consider Table 1 and its attribute *color* which should be indexed. We have 4 distinct colors and map them to the integers $0, 1, 2, 3$ as in Fig. 2. The buckets contains linked lists with pointers to the tuples in the dataset. If we want to evaluate the preference $P :=$ LAYERED(color, $(L_1 := \{\text{blue}, \text{red}\}$, $L_2 := \{\text{green}\}))$ we do a lookup for *blue* in the bucket directory ($h(blue) = 1$) and follow the pointers to the two objects with ID 3 and 4. Afterwards we search for *red* in the Hash index and find object 8. Since we already have perfect matches we do not have to consider L_2 anymore.



1	BMW	green	180	35000
2	Audi	green	170	32000
3	Mercedes	blue	200	38000
4	BMW	blue	230	34000
5	Mercedes	black	250	20000
6	Mitsubishi	black	120	50000
7	Mitsubishi	black	140	53000
8	Audi	red	150	37000

Fig. 2. Hash index. $B = 4$.



1	BMW	green	180	35000
2	Audi	green	170	32000
3	Mercedes	blue	200	38000
4	BMW	blue	230	34000
5	Mercedes	black	250	20000
6	Mitsubishi	black	120	50000
7	Mitsubishi	black	140	53000
8	Audi	red	150	37000

Fig. 3. Compressed Trie index.

Since the evaluation of LAYERED$_m$ is just to search the index for the values in L_1, and if none is found search for values in L_2, and so on, we get a *worst-case search complexity* of $|\bigcup_{i=1}^m L_i| \cdot \mathcal{O}(1)$ (all L_i sets must be searched in the worst-case). Note that in general we have $|\bigcup_{i=1}^m L_i| \ll n$ (n the size of the dataset) and therefore we speed-up the evaluation of a LAYERED$_m$ preference and all its sub-constructors.

Trie Index. The biggest use of tries is in *exact string* retrieval and hence are suitable for indexing data for LAYERED$_m$ preference queries. When a string is queried, the path from the root is traversed by looking up the characters in the query successively. If any character is absent, the query returns no answer. If the search ends in a final node (double circle in Fig. 3), the corresponding string is returned. For a query string of length l, the search finishes in $\mathcal{O}(l)$ time [2].

For a dataset of k distinct strings, a binary search tree requires $\mathcal{O}(\log_2(k))$ time to search a string. For large databases, l is much smaller than $\log_2(k)$ and, therefore, a trie searches more efficiently. The worst-case complexity to search for LAYERED$_m(A, (L_1, \ldots, L_m))$ is given by $|\bigcup_{i=1}^m L_i| \cdot \mathcal{O}(l)$, since we have to lookup each attribute value in all L_i.

Example 4. We construct a trie for the attribute *color* in Table 1. Strings with the same prefix such as "blue" and "black" share the same path up to the common prefix "bl". A space-saving version of the trie is given in Fig. 3, where all the unary nodes of a trie on a path are compressed into a single node. The edges are then labeled by substrings, and not necessarily single characters.

4 Experiments

In this section we show that our index approach outperforms a "linear scan" (**LinScan**) algorithm, which is a modified BNL with linear runtime [3], by far. In all our experiments the data tuples and index structures are held in main memory. This reduction of I/O-operations should favor the LinScan algorithm. Note that to the best of our knowledge our work is the first one on indexing base preferences and therefore there are no other index structures as competitors.

We implemented a Java 7.0 prototype, which is available as open source project on GitHub[1]. The experiments were performed on a common PC running Linux on an Intel Core1 Duo CPU with 3.33 GHz and 4 GB main memory.

[1] https://github.com/endresma/PreferenceIndex.git.

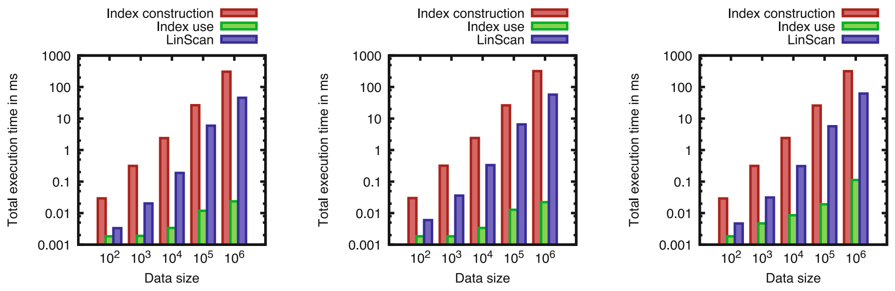
For our synthetic datasets we used the data generator commonly used in preference research [3]. For the experiments on real-world data, we used the well-known Internet Movie Database (IMDb, http://www.imdb.com), which contains information about movies. All our experiments were performed 100 times. From these measurements we took the mid half of the sorted data and use the arithmetic mean in our charts.

4.1 Numerical Index Structures

For numerical base preferences we used the Range tree index implementation as described in Sect. 3.1. All measurements on the Range tree index have been carried out with a BETWEEN$_d$ preference as it can substitute all other numerical preferences. In addition, since our implementation of the Range tree holds references to the first and last element in the tree a lookup can be skipped for LOWEST$_d$, HIGHEST$_d$, ATMOST$_d$, and ATLEAST$_d$. Hence, we do not benchmark these preferences.

Synthetic Data – Gaussian Distribution. The following experiments use generated data on a Gaussian distribution. This is common in database experiments and allows to carefully explore the behavior of index methods. Each set of data contains a number of values between 0 and 1000 (the *range* of a data set).

Figure 4 shows the measured *execution times* in nanoseconds for BETWEEN$_d$. We varied the data size (number of tuples) from 10^2 to 10^6, the d parameter ($d = 0, 10$) and the interval borders *low* and *up*. Figure 4a and b model a "real" BETWEEN$_d$ preference, whereas Fig. 4c corresponds to an AROUND$_d$ preference. Note that we use a log scale for the y-axis. It is apparent, that the time used to build the index is multiple times that needed to evaluate the preferences with the LinScan algorithm. However, if the index is constructed, the index based evaluation of the preferences is hardly noteworthy.



(a) d=0, low=0, up=100 (b) d=10, low=0, up=100 (c) d=10, low=500, up=500

Fig. 4. Execution time (in ms) for the evaluation of BETWEEN$_d(A; [low, up])$.

Figure 5 shows the maximal relative difference w.r.t. the execution time (exTime) of LinScan and the index-based evaluation of BETWEEN$_d$ preferences with varying intervals $[low; up]$ on $d = 0$ and $d = 10$. That means, we compute *deltaTime := LinScan exTime - Index exTime* and plot the ratio

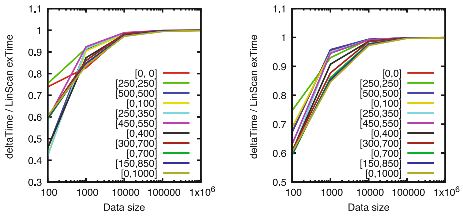
$$\gamma_1 := \text{deltaTime/LinScan exTime} \tag{1}$$

on the y-axis. This shows that with increasing data size the difference between the LinScan execution time and index execution time increases as well. Our measurements suggest that the index is more than two times faster than the LinScan even on small relations and as expected much faster for large data sets.

In Fig. 6 we present the ratio of the execution time of LinScan and the index-based preference evaluation on different $[low, up]$ intervals, i.e.,

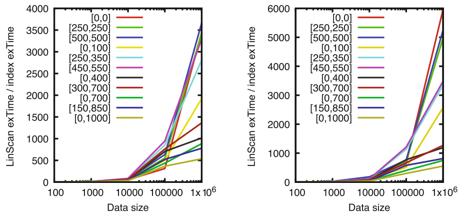
$$\gamma_2 := \text{LinScan exTime/Index exTime} \tag{2}$$

Thereby, the first three measurements have the same low and up parameters, simulating an AROUND$_d$ preference. It is probable that these sets have been the fastest because of the relatively smaller size of the BMO-set. In summary, the index-based approach is hundreds to thousands of times faster on a dataset with only 10^6 objects.



(a) d=0 (b) d=10

Fig. 5. γ_1 for BETWEEN$_d$.



(a) d=0 (b) d=10

Fig. 6. γ_2 for BETWEEN$_d$.

Influence of the Number of Distinct Values. Both the trees height and the size of the doubly linked list are determined by the number of distinct values within the dataset. The ratio between the data size and the number of distinct values is a significant factor for the effectiveness of an index. We use different sets of generated data to measure the behavior of our index techniques. Each set of data contains 10^6 objects with a varying number of distinct values from 1 to 10^6.

As the operation for inserting a value into the Range tree index is costly we expected the index build time to rise proportionally to the number of distinct values in the dataset. This has been measured and can be seen in Fig. 7. The LinScan execution time is nearly constant because of the linear scan over 10^6 tuples each time. The index execution time is much better than LinScan even for 10^6 distinct values. Again, a log scale is used for the y-axis.

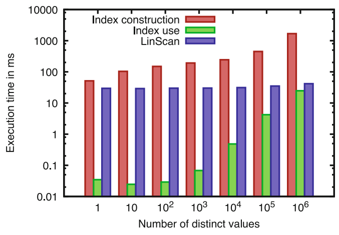


Fig. 7. Runtime for $\text{BETWEEN}_d(A, [0, 10^6])$.

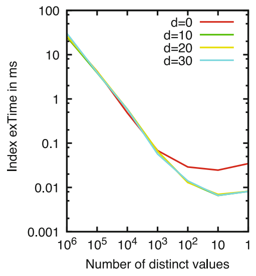


Fig. 8. Execution time.

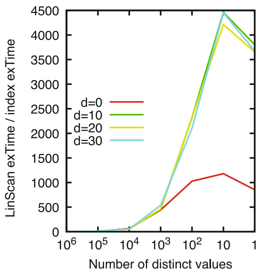


Fig. 9. γ_2

Similarly the index performs worse if there are only distinct values, cp. Fig. 8, where we used 10^6 tuples, varied the d value and the number of distinct values. Nevertheless, the index still performs better than the LinScan algorithm.

Two interesting observations can be made with Fig. 9. Firstly, the evaluation of a BETWEEN_d preference that has a parameter $d > 0$ is more costly than without. Thus the gain when using an index is greater as there are fewer evaluations needed. Secondly, a puzzling dip in the graph can be observed once there is only a single unique value. The LinScan might be relatively more efficient in this instance, because it does not need to empty its list of BMO candidates during the evaluation.

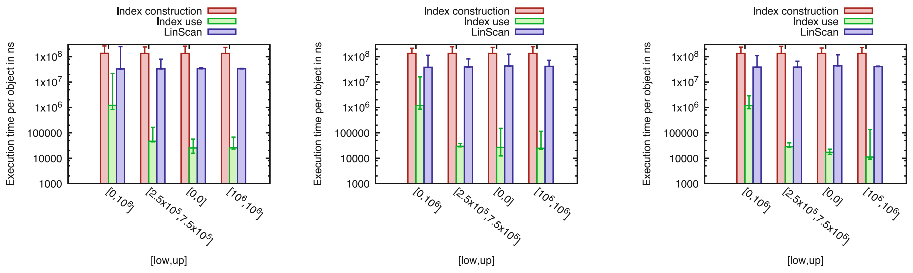
Real-World Data. The following measurements use a set of voting data from IMDb as its basis. It contains 468097 tuples and 11295 distinct values.

Figure 10 shows that building the index is more costly than an evaluation with LinScan. Again, once the index is constructed, the evaluation is much faster than LinScan. In this experiment we used $d = 0, 10, 50$, varied the BETWEEN_d preference, and plotted the *execution time per object* in ns.

Figure 11 presents the γ_1 ratio (Eq. 1) and that the evaluation on the index is much faster than with LinScan. We used different $[low; up]$ intervals for the BETWEEN_d preference and varied the d from 0 to 50.

In Fig. 12 we present γ_2 (Eq. 2) and present the ratio of the LinScan execution time to the index execution time with different $[low; up]$ intervals and d values. Again, the index-based evaluation is more costly for a d parameter $d > 0$, but still better than LinScan.

In all our real world experiments the observations made earlier on synthetic data were confirmed. Hence, our index-based evaluation of numerical base preferences is also applicable for real data sets.



(a) d=0 (b) d=10 (c) d=50

Fig. 10. Execution times on the IMDb real world data set.

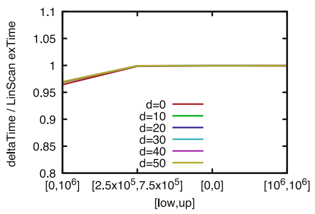


Fig. 11. γ_1

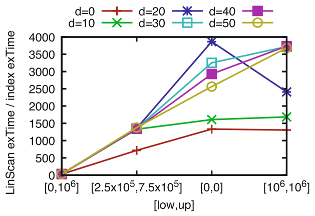


Fig. 12. γ_2

4.2 Categorical Index Structures

Our categorical preference LAYERED$_m$ was analyzed on real-world data only, because there is no useful and reasonable data generator for strings. The following measurements use a set of genre data from IMDb as its basis. It contains 1580880 objects and 31 distinct values. In the measurements we used the preference LAYERED$_2(A, (L_1, L_2))$, which is a POS preference, and varied the L_1 set in its size w.r.t. the distinct values in the data set.

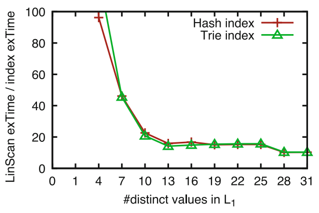


Fig. 13. γ_2 w.r.t. different sizes of L_1.

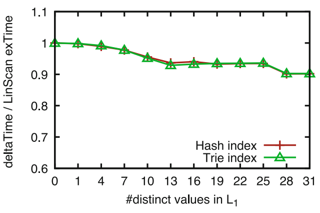


Fig. 14. γ_1 w.r.t. different sizes of L_1.

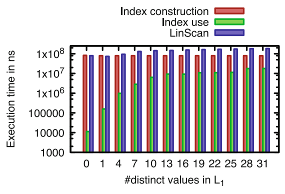


Fig. 15. Hash index.

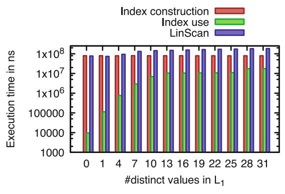


Fig. 16. Trie index.

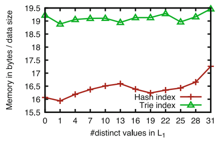


Fig. 17. Memory usage.

We compared the Hash index and the Trie index to LinScan and to each other. The Hash index implementation is based on the standard Java `HashMap` containing a corresponding list of object references for each distinct value. The Trie index implementation consists of a Prefix-Tree. Each node holds a list of objects corresponding to a unique value and holds an array for child nodes. These arrays facilitate a fast traversal of the tree.

In Fig. 13 we present the ratio of the execution time of LinScan and the index-based evaluation, i.e., γ_2 as in Eq. 2. We varied the size of the L_1 set from 0 to 31 values in our LAYERED$_2(A, (L_1, L_2))$ preference. Keep in mind, that we have 31 distinct values in our dataset and hence we retrieve the complete dataset in the case of $|L_1| = 31$ and only a fraction of the set if $|L_1| \to 0$. We skipped the results for $|L_1| = 0, 1$, because the ratio is too high for a usable presentation (actually it is about 8000). Our figure states that the index is extremely fast for small L_1 sets and still much better than LinScan if we have to check all values.

Figure 14 presents the γ_1 ratio (Eq. 1), i.e., the maximal relative difference in the execution time between LinScan and the index-based approaches. The x-axis denotes how many of the distinct values are contained in layer L_1. It shows that the index performs best for small layers and is much better than LinScan even for a layer containing all distinct values. The two index structures Hash index and Trie index perform very similar.

Something more interesting can be observed for the Hash index in Fig. 15. Depending on the layer L_1 size, the index build time is less than the LinScan execution time. In extreme cases even the combined time of building the index structure and an evaluation on it, can be faster than LinScan. Again, the Trie index performs very similar to the Hash index as can be seen in Fig. 16. In both cases the index construction time is nearly the same for all sizes of L_1.

Figure 17 represents the memory usage in the relationship to the data size of the IMDb dataset. Since our implementation is based on Java 7.0, the memory usage includes all information on all data structures, objects, references to the objects, additional memory requirements for the Java engine, ...[2] As one can see, the Hash index needs much less memory than the Trie index, but both have a similar execution time behavior.

[2] We used the Java Runtime object with the methods `totalMemory()` and `freeMemory()` to determine the total amount of used memory in the JVM.

5 Conclusion and Future Work

In this paper we presented index structures for preference database queries. Our indexing techniques rely on common database indexing approaches and therefore do not require additional adaption of a database back-end engine. One advantage of our approach is that we can provide the BMO points progressively, since index structures support this in a natural way.

Our extensive performance study shows that the proposed index methods provide quick response times compared to a linear scan when the index is build once. Hence, as with all indexes, indexing make sense with quite static data. In addition, since we rely on common index structures, building and maintaining costs for our preference indexes remain unchanged.

For future work we plan to develop indexing methods for ordered importance of preferences, i.e., Prioritization as well as geo-spatial preferences and preferences based on full text.

References

1. Bayer, R., Unterauer, K.: Prefix B-trees. ACM Trans. Database Syst. (TODS) **2**(1), 11–26 (1977)
2. Bhattacharya, A.: Fundamentals of Database Indexing and Searching. CRC Press, Chapman & Hall Book, Boca Raton (2015)
3. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proceedings of ICDE 2001, pp. 421–430. IEEE, Washington, DC (2001)
4. Chaudhuri, S., Dalvi, N., Kaushik, R.: Robust cardinality and cost estimation for skyline operator. In: Proceedings of ICDE 2006, p. 64. IEEE Computer Society, Washington, DC (2006)
5. Chomicki, J., Ciaccia, P., Meneghetti, N.: Skyline queries, front and back. In: Proceedings of SIGMOD 2013, vol. 42, no. 3, pp. 6–18 (2013)
6. Kießling, W.: Foundations of preferences in database systems. In: Proceedings of VLDB 2002, pp. 311–322. VLDB, Hong Kong, China (2002)
7. Kießling, W., Endres, M., Wenzel, F.: The preference SQL system - an overview. Bull. Tech. Commitee Data Eng. **34**(2), 11–18 (2011). IEEE Computer Society
8. Lueker, G.S.: A data structure for orthogonal range queries. In: Proceedings of FOCS 1978, SFCS 1978, pp. 28–34. IEEE CS, Washington, DC (1978)
9. Mandl, S., Kozachuk, O., Endres, M., Kießling, W.: Preference analytics in EXASolution. In: Proceedings of BTW 2015 (2015)