Pack Indexing for Time-Constrained In-Memory Query Processing

Tobias Jaekel, Hannes Voigt, Thomas Kissinger, Wolfgang Lehner

Department of Computer Science
Dresden University of Technology
D-01062 Dresden
{firstname.lastname}@tu-dresden.de

Abstract: Main memory databases management systems are used more often and in a wide spread of application scenarios. To take significant advantage of the main memory read performance, most techniques known from traditional disk-centric database systems have to be adapted and re-designed. In the field of indexing, many main-memory-optimized index structures have been proposed. Most of these works aim at primary indexing. Secondary indexes are rarely considered in the context of main memory databases. Either query performance is sufficiently good without secondary indexing or main memory is a resource too scarce to invest in huge secondary indexes. A more subtle trade between benefit and costs of secondary indexing has not been considered so far.

In this paper we present Pack Indexing, a secondary indexing technique for main memory databases that allows a precise trade-off between the benefit in query execution time gained with a secondary index and main memory invested for that index. Compared to traditional indexing, Pack Indexing achieves this by varying the granularity of indexing. We discuss the Pack Indexing concept in detail and describe how the concept can be implemented. To demonstrate the usefulness and the effectiveness of our approach, we present several experiments with different datasets.

1 Introduction

The field of database management systems (DBMS) is changing; main memory database management systems (MMDBMS) become more important. Today, main memory is as large and cost-efficient as never before. To avoid expensive I/O operations on data, MMDBMS store entire databases in-memory. The advantage of MMDBS compared to disk-based DBMS is the significantly higher read performance and that the data can be processed much more efficiently. The trend towards MMDBS enables novel application scenarios such as live business intelligence [Pla09].

To fully exploit the advantage of main memory technology, most of the well-researched technologies of disk-based DBMS have to be rethought and redesigned for MMDBMS. For instance, storing tuples row-wise is less suitable for main memory since data compression is less efficient, sequential main memory accesses are more cache-efficient and the memory's bandwidth capabilities are exploited. In consequent, most MMDBMS are based on column-

wise data storage [CK85, ADHS01]. Other techniques such as access methods, page layouts, etc. are affected equally. In the area of indexing, a lot of indexing structures were invented to use the capabilities of main memory more efficiently [KCS+10, RR00, KSHL12].

Auxiliary indexing, however, is an often neglected topic, since table scans and primary indexes access are sufficient fast on main memory in the most cases. Nevertheless, indexing becomes important when queries have to be processed within a time constraint and a scan would exceed this constraint. Today, data grows rapidly and the need of secondary indexes will become more likely to keep up with the promises of main memory databases. Indexes consume main memory, which is in contrast to disk scarce resource. Usually, relations consist of columns that contain a lot of duplicate values. These duplicates can be compressed efficiently, but the index' size stays high caused by insufficient duplicate handling. Consequently, secondary indexes cannot be created as excessively in main memory as on disk. Indexing in MMDBMS requires careful decisions about which data to index.

Secondary indexes are created because of insufficient execution times. This implies concrete objectives about the targeted execution times. Traditional indexing techniques only allow coarse-grained indexing decisions: either a column is fully indexed or not indexed at all. Ideally, the database administrator is able to tailor the index for his requirements and aimed execution times. Especially, with query time constraints the database administrator wants to spend only as much memory as necessary on indexing to meet the time constraint.

In this paper, we present Pack Indexing for main memory databases management systems. Pack Indexing is able to (1) align the execution time to a time constraint and (2) limit the memory consumption for auxiliary indexing significantly. Further, Pack Indexing enables trading query execution time for memory consumption in a fine-grained way. Aligned to a given time constraint, our novel concept indexes packs of records instead of individual records to consume as little memory as possible. To meet the time constraint for every queried value the pack size can be configured for each value individually. We also discuss the implementation of the Pack Indexing concept and present a detailed evaluation of the concepts. Pack Indexing is applicable for row-oriented as well as column-oriented storages.

The rest of the paper is structured as follows: In the following section we present the concept of Pack Indexing. Section 3 gives an overview of our implementation. The Performance Rating component is described in Section 4. In Section 5 the Pack Configurator is discussed. How the Pack Index is used during runtime is described in Section 6. An Evaluation of the Pack Index system is given in Section 7. In Section 8 we discuss the related work and in Section 9 this paper is concluded.

2 Pack Indexing

A time-constrained system has an inherent characteristic: It does its work within a given time limit. In a database system the execution time highly depends on the quantity of read tuples. Reducing the number of read tuples means decreasing the execution time. An index scan reduces the number of read tuples to the expense of memory; A column scan reads all

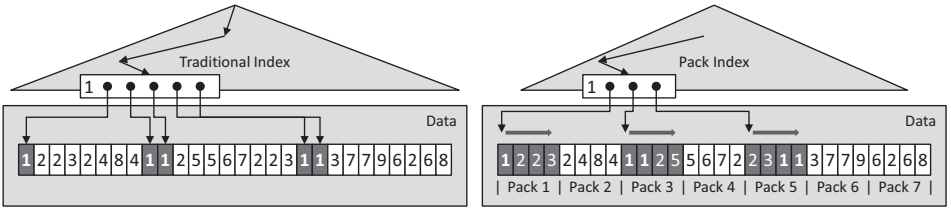

Figure 1: Traditional Index compared to Pack Index.

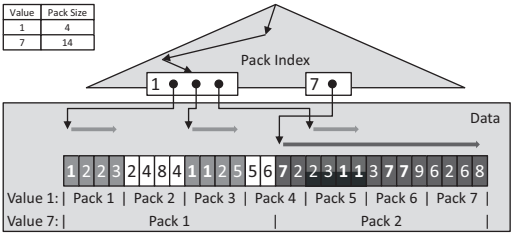

Figure 2: Pack Index with individualized pack sizes.

tuples without memory expenses. Granularity-wise, an index scan is the most fine-granular access and a column scan the most coarse-grained access.

Pack Indexing allows controlling the granularity of indexing. It provides an access method that can be configured for any granularity between a traditional index scan and a column scan. A more coarse-grained index consumes less memory than a more fine-grained index. Thus, Pack Indexing provides a mean to the trade between execution time and memory consumption.

Pack Indexing builds on two ideas: (1) Indexing packs of records instead of individual records and (2) Configuring the size of the packs for each value in the index individually depending on the value's frequency.

Figure 1 shows the first idea in contrast to traditional indexing. The example shows a column of 28 tuples and 9 distinct values. Consider Value 1, which occurs five times in the dataset. The traditional index indexes the tuples directly. This requires five index entries and allows reading the five qualifying tuples directly. In contrast, the Pack Index logically combines the tuples to packs and indexes the packs. Since Value 1 occurs multiple times in Pack 3 and Pack 5, the Pack Index needs merely three index entries. Reading the qualifying tuples, though, requires reading three complete packs with a total of 12 tuples. Assuming the targeted time constraint is at 50% of a complete scan, the Pack Index achieves the goal with two entries less than the traditional index. Where saving space is more important than the last pinch of execution time benefits, indexing packs of values allows saving index space in a controlled and directed way.

Figure 2 illustrates the second idea of Pack Indexing: Individual pack sizes for each value.

As we have seen, a pack size of four is sufficient to push queries on Value 1 below the targeted time constraint. For queries on Value 7, occurring only three times in the dataset, a pack size of 14 is a far better choice. The read costs of 14 tuples meet the assumed 50% time constraint with a single index entry. A pack size of 4 would push queries on Value 7 below the time constraint at a price of two index entries. Pack sizes individualized depending on a value's frequency allow further space savings by tailoring a Pack Index to the value distribution of a dataset.

The value distribution of a dataset is what primarily determines the concept's potential benefit. The benefit of the Pack Index approach is the amount of memory it can save for a given execution time constraint compared to a traditional index. Depending on the data distribution, the amount varies from dataset to dataset. Two kinds of distributions are to consider. The logical value distribution represents the frequencies with which the individual instances of a value domain occur in the dataset. The physical distribution denotes the physical clustering of the values on the storage. Logical value distribution and physical value distribution are orthogonal properties of a dataset and both influence the benefit of a Pack Index.

Generally, Pack Indexing aims to exploit disparities in the data distribution. Physically as well as logically uniformly distributed data has no disparities to exploit and constitutes the worst case with the lowest expectable benefit. Physical distribution has a stronger influence on the benefit than logical data distribution. If a value is uniformly distributed over the physical representation of a dataset the probability that matching records occur in a pack is equal for all packs independently from the pack size. In contrast, non-uniformly distributed values will cluster in a fraction of the packs and reduce the number of required index entries. The logical data distribution is what pack sizes are tailored to. Hence, logically uniformly distributed data, i.e., where each value occurs with the same frequency, will result in equal pack sizes for all values. The actual benefit a non-uniformly distributed value range allows, depends on the physical distribution and the index data structure the Pack Index builds on.

Summarizing, at best data is logically as well as physically non-uniformly distributed. Real-world data distribution is generally time-dependent cause by seasons, day-and-night cycle, and trends in development and style. Shopping for instance: Without doubt, you buy different things in the morning than in the evening, different things in summer than in winter, and different things in five years from now. The today increasingly common append-only databases reflect these variations in data distribution logically as well as physically. Thus, the best case for Pack Indexing is a likely case.

3 System Overview

The Pack Index system consists of three components: (1) A rating of the database systems memory read performance, (2) the Pack Configuration, and (3) the Pack Index Access Path. Figure 3 shows the three components and their basic interaction.

The Performance Rating component provides measures of the read performance of the database system. While the database system is setup – either initially or after hardware

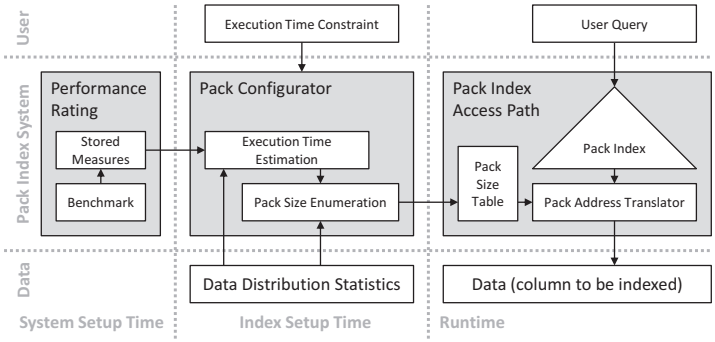

Figure 3: Pack Index System

reconfiguration – the Performance Rating component benchmarks the read performance. It stores the results of this measurement in the database catalog for the Pack Configurator to use. We detail the Performance Rating component in Section 4.

The Pack Configurator determines the individual pack sizes, if the user creates a new Pack Index. With the index create statement, the user provides the execution time constraint the index should achieve. Based on data distribution statistics taken from the system catalog, the configurator determines an individual pack size for each of distinct values in the column the index is created on. The determined pack sizes are stored in the Pack Size Table of the newly created index. We present the Pack Configurator in detail in Section 5.

The Pack Index Access Path is used during query execution. It consists of the Pack Size Table, the Pack Index, and the Pack Address Translator. The Pack Index holds the actual index entries. For a given query, it provides the pack numbers of the packs that contain matching tuples. The Pack Size Table lists the individual pack size for each value. The Pack Address Translator uses this information to translate pack numbers into memory addresses during queries and memory addresses into pack numbers during DML operations. We elaborate on the Pack Index in Section 6.

4 Performance Rating

Execution time of selection queries mainly depends on (1) how much data has to be read to find and fetch qualifying tuples and (2) how fast data can be read from memory. The Performance Rating component provides reliable measures for the memory read performance. During system setup, the component measure the read performance for a variance of read patterns and stores them in the system catalog. The Pack Configurator later uses the measures while determining individual pack sizes.

When using the Pack Index Access Path, the fetch operator has to scan a certain number of packs of a certain size. Both, number of packs and pack size influence the read performance.

Reading a small number of very large packs results in mainly sequentially memory access, while reading a large number of small packs leads to a predominantly random memory access.

The Performance Rating component captures this variance in memory read performance. It conducts a number of measurements while varying the read patter. For each read pattern the component measures read performance for a specific pack size and a specific probability that a pack has to be scanned. This results in a two-dimensional matrix of measures. Reading packs of 4 MB with a probability of 25% took 3.798 ms per pack, for instance.

To keep the number of measures in a reasonable range the Performance Rating component varies read patters logarithmically. This keeps the costs of conducting the measurements and storing the result low, while sufficiently capturing the variance in the system's read performance. To get a measure for a read pattern that has not been measured, the Pack Configurator interpolate between the two closest patterns measured. Note that the measures do not have to be stored in memory since they are only used by the Pack Configurator during index setup time. Further, the performance rating is system-dependent and is shared among all indexes in a system.

5 The Pack Configurator

The Pack Configurator determines the individual pack sizes during index creation. While creating a Pack Index the user gives the index an execution time constraint. For each value in the Pack Index, the configurator aims to find the largest pack size so that selection queries on that index fulfill the constraint. Therefore, the configurator enumerates difference pack sizes and estimates the resulting query execution time. The configuration closest to the constraint is selected.

5.1 Configuration Enumeration

Naïvely, the configurator investigates for each value every possible pack size. This guarantees to consider all possible configurations at cubic time complexity. Assuming N bytes of data and M distinct values, the configuration would require $O(M \cdot N/l)$ iterations, where l is the size of a tuple. For instance, given column store with an integer column of 1 GB containing 10,000 distinct values and an investigation speed of 10,000 iterations per second, the enumeration would last about 100 months.

We employ a more sophisticated enumeration strategy. It extends the naïve enumeration in two ways. First, we group values of the same frequency. The runtime estimation depends not on the value itself but on its frequency. Consequently, values with the same frequency result in the same pack size and we need to run the enumeration only once for each frequency. Second, we use binary search to find the largest fitting pack size for a given frequency. This reduces the number of considered pack sizes significantly to $O(M \cdot \log_2 N/l)$. In the example, the improved enumeration takes about 30 seconds to find the same configuration.

Algorithm 1 Pack Size Enumeration

```
1:  procedure ENUMERATION(f, t_c, n)
2:      ▷ f: value frequency   ▷ t_c: time constraint   ▷ n: number of tuples
3:      p_l ← 0, p_h ← log_2(n)        ▷ exponent of lowest and highest pack size
4:      c ← true
5:      t_l ← ESTIMATE(2^{p_l}, f, n)                  ▷ traditional index scan time
6:      t_h ← ESTIMATE(2^{p_h}, f, n)                  ▷ column scan time
7:      while c ∧ p_l < p_h do
8:          p ← p_l + (p_h − p_l)/2                    ▷ exponent of pack size to try
9:          s ← round(2^p)                             ▷ pack size to try
10:         t_e ← ESTIMATE(s, f, n)                    ▷ estimate execution time (section 5.2)
11:         if t_c − t_e < 0 then
12:             p_h ← p                                ▷ try smaller pack size
13:             if t_e − t_l < ϵ then                  ▷ close to lowest pack size
14:                 c ← false                          ▷ stop search
15:                 s ← 2^{p_l}                        ▷ use lowest pack size
16:         else if t_c − t_e > ϵ then
17:             p_l ← p                                ▷ try larger pack size
18:             if t_h − t_e < ϵ then                  ▷ close to highest pack size
19:                 c ← false                          ▷ stop search
20:                 s ← 2^{p_h}                        ▷ use highest pack size
21:         else                                       ▷ close to constraint
22:             c ← false                              ▷ stop search
23:     return s
```

Algorithm 1 shows the enumeration. For a given value frequency, a execution time constraint, and a total number of tuples the algorithm searches the largest pack size (in number of tuples) within the execution time constraint. Following the principle of binary search, the enumeration iteratively tests pack size configuration (line 7–22). If the estimated execution time of tested configuration exceed the given constraint, the algorithm reduces the pack size (line 11f). If the estimated execution time falls significantly below the constraint, it increases the pack size (line 16f). If the estimated time is in a small range ϵ below the constraint, the algorithm stops the search (line 21f).

In contrast to normal binary search, the enumeration increase and decrease the pack size exponentially (line 8f) instead of linearly. This follows the observation that large packs are exponentially more likely to contain a given value than small partitions, which renders large packs exponentially more unlikely to allow execution time below a given constraint. Note that exponent p is not an integer, so that the enumeration still can find any pack size. To avoid a large number of iterations towards the end of search without any reasonable change in the outcome, the algorithm stops if it gets ϵ-close to either the constraint, the execution time of the smallest possible pack size, or the execution time of the largest possible pack size. Thereby, ϵ controls the accuracy of the search.

5.2 Execution Time Estimation

To evaluate pack size configurations, the configurator estimates the execution time of a selection query given a certain pack size, the frequency of the queried value, and the total number of tuples. Essentially, query execution with the Pack Index Access Path consists of reading the Pack Index and fetching the matching tuples.

The index access consists of finding the matching entries and reading these entries. Most of the common index structures find entries in constant time C. Whereas reading the entries generally depends on the number of entries to read, specifically the number of matching packs. Hence, index access time is

$$t_{IX}(s, f) = C + r(size(n_p(s, f)))$$

where $n_p(s, f)$ is the number of matching packs, $size$ gives the size of the corresponding index entries in bytes, and r is the performance rating for a single chunk of the given size. Note that $size$ depends on the specific index structure chosen for the Pack Index.

The fetch operation has to read all matching packs. Accordingly, the fetch time is

$$t_F(s, f) = n_p(s, f) \cdot r(n_p(s, f) \cdot l, f)$$

where $n_p(s, f)$, again, is the number of matching packs, l is the tuple length in bytes, and r yields the performance rating for packs of the given size and at the given read probability.

To estimate the number of matching packs, we adopt the page fetch rate of row-level Bernoulli sampling given in [HK04]. In our case, with a total of n tuples, the expected number of matching packs for a pack size of s tuples and value frequency f is

$$n_p(s, f) = \frac{n}{s} \cdot (1 - (1 - f)^s) \quad .$$

This assumes physically uniformly distributed data. Clustered data would result in a smaller number of packs that would have to be read effectively.

6 The Pack Index Access Path

The Pack Index Access Path is used to answer queries at system runtime. It consists of the Pack Size Table, the Pack Index, and the Pack Address Translator.

The Pack Size Table contains the individual pack sizes of the index values. Formally, it is a function $\mathcal{S} : D \rightarrow \mathbb{N}$, which maps the indexed value domain to a natural number representing the pack size in number of tuples per pack. It can be implemented as a separated lookup structure or can be integrated with the Pack Index.

The Pack Index holds the actual index entries. Formally, the Pack Index is a function $\mathcal{I} : D \rightarrow \mathbb{N}^*$, which maps the indexed value domain to a list of pack numbers. Additionally, we assume the Pack Index returns matching pack numbers in ascending order. Most of the

common indexing data structures such as B-Trees, Hashing, or Bitmap Indexes can be used to implement the Pack Index.

The Pack Address Translator calculates memory addresses from pack numbers and vice versa. It realizes two functions: (1) Forward translation turns a pack number i and a pack size s into a memory address a with $a = \mathcal{F}(i, s) = s \cdot i \cdot l$. (2) Backward translation turns a memory address a and a pack size s into a pack number i with $i = \mathcal{B}(a, s) = a/(s \cdot l)$. Here, l is the tuple byte length. Query processing uses forward translation, while DML operations that have to update the Pack Index require backward translation. Addresses are relative to the beginning of the column.

The Pack Index Access Path answers selection queries with point predicates, range predicates, or predicate disjunctions. Naturally, the data structure used for the Pack Index has to support these kinds of predicates, too. Given a predicate, the Pack Index Access Path retrieves matching pack numbers from the Pack Index, translates the pack numbers to memory addresses and scans the corresponding packs to fetch matching tuples.

Point predicates are the simple case since they query a single value and all pack numbers returned by the Pack Index belong to the same pack size. Algorithm 2 shows the basic procedure. The Pack Index Access Path simply iterates the pack numbers returned by the Pack Index (line 3), forward translates the pack numbers, and scans the corresponding tuples (line 4). Because the Pack Index returns pack numbers in ascending order, the procedure scans the memory as sequentially as possible, hopping only where it has to omit not matching packs.

Algorithm 2 Answering Point Predicates

```
1: procedure POINTPREDICATEQUERY(v)                ▷ v: queried value
2:     s ← S(v)
3:     for all i ∈ I(v) do                          ▷ iterate pack numbers
4:         SCAN(F(i, s), s · l)                     ▷ scan pack (l: tuple byte length)
```

Processing range predicates and predicate disjunctions is more complex since matching packs may differ in their sizes. Regarding their treatment by the Pack Index Access Path, range predicates and predicate disjunctions can be generalized to set predicates. A value matches a set predicate if it is contained in a set of values given by the predicate. Naïvely, the Pack Index Access Path would process each value in the set predicate separately and unite the results of the scans. Main drawback of this approach is that tuples in overlapping packs would be read multiple times. For instance, consider the situation shown in Figure 2. If Value 1 and Value 7 are queried in the same set predicate, the naïve approach would scan the tuples in Pack 5 twice.

To avoid repeated scanning of tuples, the Pack Index Access Path has to align pack addresses and remove duplicates. Algorithm 3 shows the basic procedure. Given a set predicate, the Pack Index Access Path determines the common pack size, essentially, the greatest common divisor of the individual pack sizes of the queried values (line 2). Then it aligns for all queried values the pack numbers returned by the Pack Index to the common pack size (line 3–9). This can be done easily by forward translate the pack number with its corresponding pack size and backward translate the resulting address with the common

pack size (line 6). Unfortunately, the backwards translation returns only the address of the first pack with the common pack size. All following packs have to be added manually by the algorithm (line 8). This is done by adding as much packs as common pack size would fit into the original pack size subtracted 1, because the first pack number was created by the backwards translation. After aligning the pack numbers, the Pack Index Access Path merges the individual pack number lists into a single list. Thereby, it keeps the order of the pack numbers and removes duplicates. All of which can be done efficiently with the standard merge sort procedure (line 10). Finally, the Pack Index Access Path iterates the merged list of aligned pack numbers (line 11), forward translates the pack numbers with the common pack size, and scans the corresponding tuples (line 12).

Algorithm 3 Answering Set Predicates

```
1:  procedure SETPREDICATEQUERY(V)                          ▷ V: set of queried values
2:      s_c ← gcd(S(V))                  ▷ greates common divider of all matching pack sizes
3:      for all v ∈ V do                                     ▷ iterate queried values
4:          s ← S(v), I_v ← I(v)
5:          for all i ∈ I_v do
6:              i ← B(F(i, s), s_c)                          ▷ align first pack number
7:              I_v ← I_v ∪ i
8:              for all c ∈ [1, s/s_c − 1] do                ▷ add following pack numbers
9:                  I_v ← I_v ∪ (i + c)
10:     I ← merge all I_v                                    ▷ merge sort pack numbers
11:     for all i ∈ I do                                     ▷ iterate pack numbers
12:         SCAN(F(i, s_c), s_c · l)                         ▷ scan pack
```

7 Evaluation

We conducted a series of experiments to evaluate the Pack Index concept. Specifically, we examined the approaches impact on query runtime and memory consumption. Further, we investigated the influence of query selectivity and data distribution. All experiments were performed on a system equipped with an Intel i5 3450 3.1GHz CPU, 16GB of DDR3 1600MHz main memory, and Windows 7 x64 Professional as operating system.

For the experiments, we prototypically implemented all Pack Indexing components for an in-memory column store. The column store uses directory compression, so that all columns store fixed-length directory keys. Read-optimized, a column is stored in one big block in memory. The directory keys are integers running from 0 to n for n distinct values. Our prototype uses bitmap indexes for the Pack Index. All bitmaps can be efficiently looked up through a pointer array that uses the directory keys as array indexes. The Pack Size Table is implemented analogously.

We used two datasets. The first dataset is the well-known Netflix dataset of movie ratings. We indexed the movie column. It contains 17,770 distinct movies for more than 100 million movie ratings which total at about 400 MB. The physical order of values remained

unchanged. We used the first dataset to investigate query runtime and index memory consumption. The second dataset is synthetic and consists of a single column containing 100 million integer values. We generated multiple versions of the dataset with varying number of distinct values and value distributions. In all versions of the second dataset, the physical order of values is random with uniform distribution. The second dataset was used to investigate the influence of query selectivity.

For all experiments, we used the same performance ratings which were measured once before the experiments. For each experiment, we created a Pack Index for a given time constraint using the Pack Index Configurator and the data distribution statistics were determined from each dataset after the load.

7.1 Query Execution Times

First, we investigated the resulting query execution times when using a Pack Index configured for varying execution time constraints. We varied the execution time constraint from 12.5% to 100% of the runtime of a column scan. For each movie m in the Netflix dataset, we measured the runtime of the point query `SELECT movie FROM Netflix WHERE movie = 'm'`.

Figure 4 shows the results. When using the Pack Index Access Path, the query runtime varies from movie to movie mainly because of the individual pack sizes configured for each movie. The distribution of query runtimes is shown in the figure by the box plot. The figure also shows the corresponding execution time constraint. As can be seen, the Pack Index Access Path did not exceed the time constraint. The only exception is the constraint of 12.5% where the three most frequent movies are too frequent to be fetched within the given constraint. Consequently, no index technique could push a query on these movies below the constraint. For comparison, the figure also shows the runtimes of a full column scan.

Further, the results show a wide spread in the execution times when using the Pack Index Access Path. Reason of this spread is the physical distribution of the values. The Pack Index Configurator assumes physically uniformly distributed data. The movies in Netflix dataset, though, are non-uniformly distributed; the ratings have chronological order and a movie is most frequently rated when it hits the Netflix' DVD racks. In consequence of the non-uniform physical distribution of the Netflix data, the Pack Index Configurator overestimates the number of packs. A smaller index without violating the execution time constraint would be possible. However, this requires detailed knowledge about the physical order of the tuples during index creation time and reduces the index' robustness against changes in the physical tuple order.

7.2 Memory Consumption

Second, we examined the memory consumption of the Pack Index configured for varying execution time constraints. Again, we varied the execution time constraint from 12.5% to

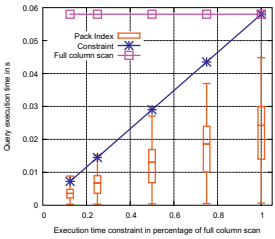

Figure 4: Query runtime of Pack Index

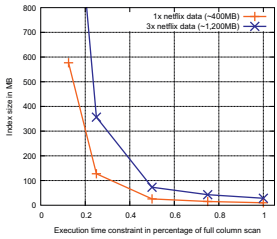

Figure 5: Memory consumption of Pack Index

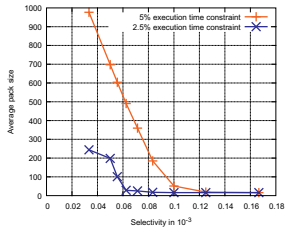

Figure 6: Selectivity of the dataset

100% of the runtime of a column scan. We used (1) the Netflix dataset in its normal size and (2) the Netflix dataset tripled in size. The tripled Netflix dataset is essentially three concatenated copies of the original dataset. For each dataset and execution time constraint, we measured the size of the Pack Index.

Figure 5 shows the results. As can be clearly seen, the size of the Pack Index decreases with an increasing execution time constraint. This clearly illustrates the trade-off between execution time and memory usage we wanted to achieve. The longer queries runtimes you are willing to accept, the less memory you have to spend on indexing. Nevertheless, the trade-off was not linear. To push all queries below 50% runtime of a full column scan, we only had to spend 26 MB which is roughly 6% of the data size. A constraint 25% already required an index as large as 29% of the data size.

The trade-off observations hold independently from the size dataset. As the figure also shows, a larger dataset exhibits the same runtime–space trade-off. However, a larger dataset requires a larger index. Provided that the logical and physical data distribution remains the same, the index grows linearly with the data.

7.3 Selectivity of the Index

The wide spread in execution times in the first experiment is caused by different query selectivities. To analyze the selectivity's impact on the time constraint, we created a synthetic dataset with 100 million tuples and varied the selectivity. With physically and logically randomly distributed data, the number of distinct values specifies the selectivity of an index. We investigated different selectivities in the range from $0.033 \cdot 10^{-3}$ (30,000 distinct values) up to $0.166 \cdot 10^{-3}$ (6,000 distinct values). These test were made for two different time constraints, 2.5% and 5% of the column scan time. For each selectivity we measured the average pack size.

Figure 6 shows the results. As you can see, with a higher selectivity, which means the column holds more distinct values for the same total number of tuples, a bigger average pack size is used. In contrast, a lower selectivity leads to smaller packs, which consume more memory. Furthermore, Figure 6 illustrates the selectivity's and time constraint's impact on the pack size. For an average pack size of 200 tuples per pack, the test with

the 2.5% time constraint requires a selectivity of $0.05 \cdot 10^{-3}$ where a system with 5% only need a selectivity of $0.08 \cdot 10^{-3}$. In real-world scenarios point queries have different selectivities, depending on the queried value. To maximize the memory savings and to exploit the differences in selectivity, each value has to be aligned independently.

The results also show, for highly selective datasets the time constraint can be decreased without fully indexing the data. For instance, the tests for a selectivity of $0.055 \cdot 10^{-3}$. The 5% time constraint had an average pack size of 600 tuples per pack whereas the other test had a average pack size of 100. For both examples no fully index was required to meet the given time constraint. However, the time constraint that can be achieved when using the Pack Index strongly depends on the selectivity of the queries.

8 Related Work

The Pack Index is a novel indexing concept and independent of the index structure. However, Pack Indexing is designed for main memory databases. Indexing structures suitable for characteristics of main memory were developed in the past decade [CK96, KCS+10, KSHL12]. Theses structures are highly optimized and speed up the MMDBMS by avoiding cache misses. Combining Pack Indexing with such optimized structures leads to a win-win situation. On the one hand the index structures benefit from using Pack Indexing while consuming less memory. On the other hand the capabilities of memory saving will be exploited, if Pack Indexing is used in combination with a highly optimized index structure.

Pack Indexing aims at time-constrained query execution times, a recurring goal in database research over the last decades. In [HOT89] a statistical method for aggregation queries is presented. A similar approach is used in [OGDH95] for non-aggregation queries. Both methods are based on statistical estimation and once the time for processing the query is exceeded, the evaluation process will be interrupted. In [RSQ+08] the authors present a system that constantly scans the partitions to answer queries. Thus, the system is able to answer a query within a constant time. By constantly scanning partitions, the query can never be answered faster than the scan execution time. Pack Indexing focuses on scenarios in which a faster execution time is required. At the same time the indexes have to consume as little memory as possible to meet the constraint.

Time-constrained systems are closely related to real-time systems, here real time database systems (RTDBS). An overview of methods and techniques is given [KGm95]. RTDBS deal with hard time constraints and are used in special domains such as monitoring. In monitoring scenarios a high update or insert ratio is very important. In contrast, the Pack Index is oriented towards read-intensive scenarios.

Partial Indexing is a useful approach to save memory and index maintenance costs, even in main memory databases [Sto89, SS95]. A partial index only indexes a subset of tuples which helps to keep the index small, but if an unindexed value is queried the execution time increases to the scan time. Thus, to apply partial indexes detailed information about the workload of the database system is required. Our approach also saves memory, but not by leaving tuples unindexed. Additionally the Pack Index is independent of any query

workload knowledge. Pack Indexing and partial indexing are complementary approaches.

A pack contains a subset of tuples of the table like a horizontal partition does. Partitioning is well-known and all major database vendors introduced database design advisors, which recommend index, views and partitioning configurations [ACK+04, ZRL+04, DDD+04]. In modern cloud systems partitioning is also used for scalability [AEAAD10]. In contrast to partitioning, which is applied physically, packs do not split the data into physical fragments.

9 Conclusions

The amount of data processed and stored in main memory databases grows rapidly, but the techniques of disc-centric database systems cannot transferred without further ado. In this paper we proposed Pack Indexing, a technique that is suitable for row-oriented as well as column-oriented storages. Pack Indexing builds on two ideas: (1) Indexing packs of records instead of individual records and (2) Configuring the size of the packs for each value in the index individually depending on the value's frequency. These techniques allow us to control the granularity of an index to create a trade-off between execution time and memory consumption.

We explained an implementation of Pack Indexing consisting of three components. The Performance Rating component measures the system's read performance. These results in combination with the time constraint are used by the Pack Configurator to determine the pack size for each value individually. Both, the Performance Rating and Pack Configurator are used during setup time. At runtime, the Pack Index Access Path provides the actual benefit to the queries. Additionaly, we evaluated our prototypical implementation in several experiments. Our tests showed the benefit of Pack Indexing in memory consumption while the system did not exceed the time constraint.

References

[ACK+04] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB'04*, 2004.

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, 2001.

[AEAAD10] Divyakant Agrawal, Amr El Abbadi, Shyam Antony, and Sudipto Das. Data Management Challenges in Cloud Computing Infrastructures. In *Databases in Networked Information Systems*, volume 5999 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin / Heidelberg, 2010.

[CK85] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, SIGMOD '85, 1985.

[CK96] Kong-Rim Choi and Kyung-Chang Kim. T*-tree: a main memory database index structure for real time applications. In *Real-Time Computing Systems and Applications, 1996. Proceedings., Third International Workshop on*, 1996.

[DDD+04] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. Automatic SQL Tuning in Oracle 10g. In *VLDB'04*, 2004.

[HK04] Peter J. Haas and Christian König. A bi-level Bernoulli scheme for database sampling. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, 2004.

[HOT89] Wen-Chi Hou, Gultekin Ozsoyoglu, and Baldeo K. Taneja. Processing aggregate relational queries with hard time constraints. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, 1989.

[KCS+10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD'10*, 2010.

[KGm95] Ben Kao and Hector Garcia-molina. An Overview of Real-Time Database Systems. In *Advances in Real-Time Systems*, pages 463–486. Springer-Verlag, 1995.

[KSHL12] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. KISS-Tree: smart latch-free in-memory indexing on modern architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, 2012.

[OGDH95] G. Ozsoyoglu, S. Guruswamy, Kaizheng Du, and Wen-Chi Hou. Time-constrained query processing in CASE-DB. *Knowledge and Data Engineering, IEEE Transactions on*, 7(6):865 –884, dec 1995.

[Pla09] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, 2009.

[RR00] Jun Rao and Kenneth A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *SIGMOD'00*, 2000.

[RSQ+08] V. Raman, G. Swart, Lin Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, 2008.

[SS95] P. Seshadri and A. Swami. Generalized partial indexes. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 420 –427, mar 1995.

[Sto89] M. Stonebraker. The case for partial indexes. *SIGMOD Rec.*, 18(4):4–11, December 1989.

[ZRL+04] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB'04*, 2004.