Kläbe, Steffen; Sattler, Kai-Uwe; Baumann, Stephan

PatchIndex: exploiting approximate constraints in distributed databases

Original published in:	Distributed and parallel databases. - New York, NY [u.a.] : Consultants Bureau. - 39 (2021), 3, p. 833-853.
Original published:	2021-03-06
ISSN:	1573-7578
DOI:	10.1007/s10619-021-07326-1
[*Visited:*	2022-03-03]

Distributed and Parallel Databases (2021) 39:833–853
https://doi.org/10.1007/s10619-021-07326-1

PatchIndex: exploiting approximate constraints in distributed databases

Steffen Kläbe[1] · Kai-Uwe Sattler[1] · Stephan Baumann[2]

Accepted: 22 February 2021 / Published online: 6 March 2021

Abstract
Cloud data warehouse systems lower the barrier to access data analytics. These applications often lack a database administrator and integrate data from various sources, potentially leading to data not satisfying strict constraints. Automatic schema optimization in self-managing databases is difficult in these environments without prior data cleaning steps. In this paper, we focus on constraint discovery as a subtask of schema optimization. Perfect constraints might not exist in these unclean datasets due to a small set of values violating the constraints. Therefore, we introduce the concept of a generic PatchIndex structure, which handles exceptions to given constraints and enables database systems to define these approximate constraints. We apply the concept to the environment of distributed databases, providing parallel index creation approaches and optimization techniques for parallel queries using PatchIndexes. Furthermore, we describe heuristics for automatic discovery of PatchIndex candidate columns and prove the performance benefit of using PatchIndexes in our evaluation.

Keywords Self-managing databases · Distributed databases · Schema refinement · Approximate constraints · Uniqueness · Patch processing

✉ Steffen Kläbe
steffen.klaebe@tu-ilmenau.de

Kai-Uwe Sattler
kus@tu-ilmenau.de

Stephan Baumann
stephan.baumann@actian.com

[1] TU Ilmenau, Ilmenau, Germany

[2] Actian Germany GmbH, Ilmenau, Germany

1 Introduction

In a cloud data warehouse environment self-management and ease-of-use becomes important. This is reinforced by cloud warehouse applications typically lacking a database administrator. Self-managing systems try to overcome this with automatic schema tuning to achieve reasonable query performance.

In this paper, we focus on database constraints as an important factor for high query performance. Cloud warehouse applications integrate data from different sources, potentially leading to unclean data due to e.g., the integration of heterogeneous schemas. On these datasets, the automatic discovery of constraints is more difficult, as there might be a small number of tuples violating a constraint. Consequently, constraints can not be defined, leading to poor query performance as the query optimizer is not aware of any applicable optimizations, e.g. dropping aggregations in case of uniqueness, or the choice of specialized physical operators, e.g. foreign-key joins or merge joins in case of a sorting constraint. One possible solution for this problem is data cleaning, so e.g. infering missing values according to an observed data distribution or deleting tuples that violate a constraint. For cases where data manipulation is not desired, we introduced the concept of PatchIndexes in [12] as an alternative approach, enabling database systems to define approximate constraints without modifying data. These constraints hold for all values of an indexed column except a set of exceptions maintained by the PatchIndex. As examples we introduced "nearly unique columns" (NUC) and "nearly sorted columns" (NSC).

We investigated the PublicBI benchmark [26] to prove the existence of approximate constraints and encourate the need to handle them by the DBMS. The benchmark is a collection of real world Tableau workbooks, allowing evaluations against real user datasets and their common properties, e.g., many string columns, many NULL values or the absence of constraint definitions. We chose the *USCensus_1*, the *IGlocations2_1* and the *IUBlibrary_1* workbook as examples and determined the number of columns that contain an approximate constraint and the number of tuples that match these constraints. The histogram shown in Figure 1 represents the distribution of approximate constraint columns for these datasets. The *USCensus_1* workbook contains over 500 columns, from which 15 columns match an approximate sorting constraint. Nine columns match the sorting constraint with over 60% of their tuples. The *IGlocations2_1* and the *IUBlibrary_1* workbooks contain a small

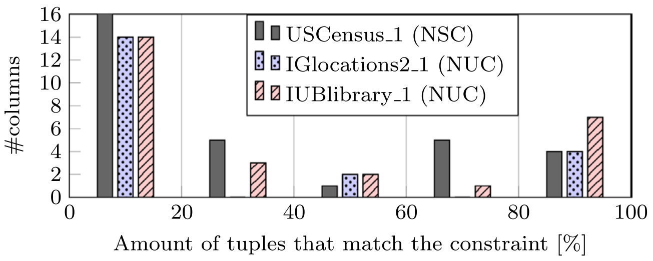

Fig. 1 Histogram over approximate constraint columns in PublicBI datasets

number of columns, from which a relatively large amount follow an approximate uniqueness constraint. Many of these columns are nearly perfectly unique.

Cloud warehouse solutions are typically designed as Massively Parallel Processing (MPP) systems, where the key for efficient query processing is to partition and distribute data within the system in order to query it in a distributed fashion. With this paper, we apply and extend the concept of PatchIndexes presented in [12] to parallel and distributed environments. Our main contributions are:

- We apply PatchIndexes to the partitioned environment.
- We provide parallel approaches for PatchIndex creation.
- We describe opportunities to optimize parallel queries using PatchIndexes.
- We present heuristics for automatic PatchIndex candidate discovery.
- As the PatchIndex structure is designed in a generic way, we discuss different opportunities where PatchIndexes can be applied in the future.

2 Related work

The research on approximate constraints evolved from the field of constraint discovery, which is typically based on data profiling techniques [1]. An example problem is the discovery of unique column combinations (UCC), which is a set of columns whose projection only contains unique rows. Finding all exact UCCs for a given relation is shown to be NP-hard [8] and various algorithms were introduced in [9, 19, 22, 23].

These algorithms mainly focus on the discovery of exact UCCs, so unclean data might hamper the discovery. In order to cope with this problem, numerous approaches and classes for data cleaning are known [21]. Alternatively, "possible" and "certain" keys [14] replace violating tuple values and enforce constraints and "embedded uniqueness constraints" (eUC) [27] separate uniqueness from completeness by enforcing uniqueness constraints only on the subset of tuples without the occurrence of NULL values. Not only taking NULL values into account, the PYRO algorithm [13] discovers and ranks approximate functional dependencies and UCCs, extending the pruning rules of the TANE algorithm [10] and using a sampling strategy for candidate pruning to reduce the search space. Recent publications [16, 20] also cover discovery approaches for approximate denial constraints, which is a more general class of data-specific constraints.

The approaches presented above mainly focus on the discovery of approximate constraints, but leave out possibilities to benefit from their definition. In our work, we integrate approximate constraints into query execution in order to accelerate query performance. We combine the concept of approximate constraints with the concept of patch processing, handling exceptions to certain distributions or properties of data, which is widely used in data compression. In order to make compression schemes more robust to outliers, PFOR, PFOR-DELTA and PDICT compression schemes were introduced in [28]. Additionally, white-box compression [6] aims at automatically learning functions or properties of the data, instead of choosing a compression scheme for all values of a column. Compression can be optimized by varying algorithms between values that follow the observed behaviour and values that are exceptions to this, leading to a significant increase of compression ratios.

3 Definitions

In this section we introduce the concepts of “nearly unique columns”(NUC) and “nearly sorted columns”(NSC) as examples of approximate constraints. Tuples of such columns satisfy the uniqueness or the sorting constraint, respectively, with nearly all column values. The tuple identifiers of the tuples violating the constraints are collected in a set of patches P_c for a given column c. Consequently, these columns contain perfect constraints after excluding the tuples of P_c. In the example in Figure 2 we can state column c as unique, if we do not consider the tuples with identifiers in, e.g., $P_c = \{1, 2, 3, 7\}$. Similarly, the column c is sorted if we exclude tuples in, e.g., $P_c = \{4, 7\}$. In the following, we provide formal definitions of the concepts to base further discussions on.

Definition 1 (*Naming conventions*)
We state the following set of naming conventions for further discussion.

Symbol	Explanation
R	Relation
$t \in R$	Tuple of relation R
$dom(c)$	Set of possible values of a column c
id	Column of tuple identifiers
$id(t) \in \mathbb{N}$	Tuple identifier of t
$c(t) \in dom(c)$	Value of column c of tuple t

Additionally, we define a projection function

$$PROJ(R, c) : relation \times column \rightarrow relation$$

as a projection of relation R on column c, which similarly to the SQL operator performs no duplicate elimination and therefore differs from the relational algebra operator π.

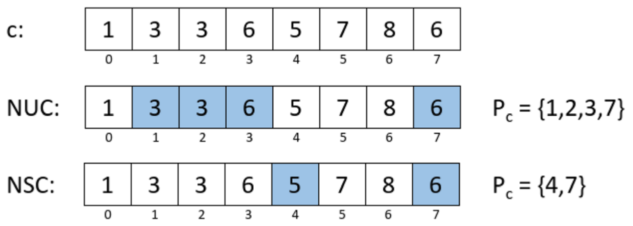

Fig. 2 Example for NUC and NSC for a given dataset

Definition 2 (*Set of patches*)
For a column c we define a set of patches $P_c \subseteq \{id(t) \mid t \in R\}$. Based on this, we define $R_P = \{t \in R \mid id(t) \in P_c\}$ as the set of tuples of R whose tuple identifiers are in P_c and $R_{\setminus P} = \{t \in R \mid id(t) \notin P_c\}$ as the set of tuples of R whose tuple identifiers are not in P_c.

Definition 3 (*Threshold variables*)
We define variables *nuc_threshold* and *nsc_threshold*, both in $[0, 1] \subset \mathbb{R}$.

Definition 4 (*Nearly unique column* (*NUC*))
A column c is a nearly unique column (NUC), when there is a set of patches P_c such that all of the following conditions are fulfilled:

(**NUC1**) $PROJ(R_{\setminus P}, c)$ is unique
(**NUC2**) $PROJ(R_{\setminus P}, c) \cap PROJ(R_P, c) = \emptyset$
(**NUC3**) $|P_c|/|R| \leq nuc\_threshold$

As an intuition, we describe values of c using the projection operator *PROJ* on c and demand these values to be unique after we excluded all tuples with tuple identifiers in P_c. The second condition (NUC2) is of major importance here to ensure the correctness of the query result, as we later want to query R_P and $R_{\setminus P}$ separately from each other during query execution. In column c of Fig. 2 values 3 and 6 are duplicates, so P_c consists of tuple identifiers of all occurences (according to (NUC2). The column c would be classified as a NUC if $nuc\_threshold \geq 0.5$. The choice of a minimal set is obviously unambiguous.

Definition 5 (*Nearly sorted column* (*NSC*))
Given a column c, let $\lhd$ be an arbitrary order relation on $dom(c)$. Column c is a nearly sorted column (NSC), when there is a set of patches P_c such that both of the following conditions are fulfilled:

(**NSC1**) $\forall t_i, t_j \in R_{\setminus P} : id(t_i) < id(t_j) \Rightarrow c(t_i) \lhd c(t_j)$
(**NSC1**) $|P_c|/|R| \leq nsc\_threshold$

As an intuition, we want values of c to be sorted according to the given order relation $\lhd$ based on the order of their tuple identifiers after we excluded tuples with tuple identifiers in P_c. For column c of Fig. 2 we can exclude the two tuple identifiers shown by P_c to get a sorted sequence, so column c could be classified as a NSC if $nsc\_threshold \geq 0.25$. This choice is ambiguous, as we can find a set of patches $P_c = \{3, 7\}$ with the same cardinality. In the discovery mechanism of NSC in Sect. 4 we are interested in a smallest set P_c.

4 Constraint discovery

Given the formal definitions of Sect. 3, classifying a given column c as a NUC or a NSC translates to the problem of finding a set of patches P_c that matches the given requirements of a NUC or NSC, respectively. In this section, we provide basic discovery approaches for both NUC and NSC that can be easily integrated into arbitrary (automatic) database administration tools. For NUC, finding non-unique values can be realized using a hash table. As this is also the core concept of hash-based aggregation operators in many database engines, we can realize the NUC discovery on query level. The main challenge here is to find all occurences of non-unique values due to the definition of (NUC2), so a simple distinct query is not sufficient for this purpose. Alternatively, we group the data on the examined key c and post-select all groups with more than two elements. The values of c that belong to the resulting groups are joined back with the table to get the tuple identifiers of all their occurences. Here we need to pay special attention to NULL values, which should be assigned to the set of patches P_c to ensure correctness. As NULL values do not join with each other, the join is realized using an outer join with a subsequent filter operations. The described approach leads to the NUC discovery query in Listing 1. Based on the result of the query, the classification of a column c as a NUC can be made based on the condition (NUC3) $|P_c|/|R| \leq nuc\_threshold$.

Listing 1: NUC discovery query

```
select tab.tid from tab left outer join
        (select c from tab group by c
        having count(*) > 1) as temp
on tab.c = temp.c
where temp.c is not null or tab.c is null
```

Finding a minimal set of patches P_c such that a given column c is sorted after excluding tuples in P_c can be realized by computing the longest sorted subsequence in c and inverting the result with respect to the base relation. As the sorted subsequence has maximum length, the respective set of patches P_c has minimal cardinality. Finding a longest sorted subsequence is a typical problem of dynamic programming and we utilize the longest sorted subsequence algorithm in [5]. The algorithm maintains arrays to keep the length and the predecessor of the last element in the longest sorted subsequence of data $[1, \dots, k]$ at position k. For each of the n elements in the array, the algorithm performs a binary search on the already computed results, resulting in an overall worst case runtime of $O(n \cdot \log(n))$. In order to compute P_c, the resulting list of indexes that are included in the longest sorted subsequence is inverted (with respect to the examined relation R). NULL values are also assigned to P_c in order to ensure correctness of sorting queries. The classification of a column c as a NSC can then be based on the condition (NSC2) $|P_c|/|R| \leq nsc\_threshold$.

5 Index design

The design of the PatchIndex data structure follows the requirement to efficiently maintain and access the set of patches P_c for the indexed column c the index is built on. The access pattern of the index scan described in Sect. 6.2 is a sorted traversal of the data and the set of patches P_c, which needs to be considered when choosing the PatchIndex data structure. Different traditional index structures could be used to match this requirement. First, the tuple identifiers in P_c could be stored in a B$^+$-Tree [7]. Although B$^+$-Trees also offer sorted traversal by providing pointers between the leaf nodes, storing and maintaining the tree structure introduces overhead while the benefit of a fast point query is not necessary for the index scan access pattern. Hash-based indexes are also not suitable for the ordered scan pattern due to the cache-unfriendly random access [7]. Bloom filters [17] allow efficient membership queries but also allow false positives. This approximate nature of bloom filters would harm the definitions presented in Sect. 3 and lead to wrong query results when applying the optimization techniques described in Sect. 6.

We realized a sparse and a dense approach of storing this information. In the identifier-based approach, which is similar to the sparse way of storing data, we hold a list of 64 Bit tuple identifiers of the tuples in P_c. On the contrary, the bitmap-based approach, which is similar to the dense way of storing, holds a single bit for each tuple, indicating whether it belongs to the set of patches or not. This is particularly independent from the cardinality of P_c. As a consequence, the main decision point between both approaches is the memory consumption. As the bitmap-based approach has a constant memory consumption of one bit per tuple, the identifier-based approach has a lower memory consumption for all cases with exception rates $e = |P_c|/|R| \leq 1/64 = 1.56\%$.

A major advantage of the approach of storing the PatchIndex information separately from the actual data is that the physical data order is not changed. Consequently, it becomes possible to define multiple PatchIndexes on the same table. This particularly enhances sorting constraints and the usual sort keys, which would physically reorder the data. Therefore, PatchIndexes offer to exploit (nearly) co-sorted columns, like orderdates and shipdates or auto-increasing identifiers. Furthermore, one can create PatchIndexes for different constraints on a single column, e.g., if it is nearly unique and nearly sorted.

The key for parallel and distributed query processing is partitioning, splitting tuples into physical chunks of the input relation. Usually this is realized using hashing on a set of partitioning attributes. Queries can then be executed in a parallel or distributed fashion by assigning partitions to cores or cluster nodes respectively. Applying the concept of PatchIndexes to the partitioned environment, we create a separate index for each partition, which has the advantage that partitioning is transparent for the actual index implementation. Additionally, PatchIndex seamlessly integrate into parallel and distributed query processing with this approach, as the separate indexes are coupled with their partitions and are therefore independent from each other. In cloud warehouse systems that rely on partitioning, work (re)distribution is done by (re)assigning partitions to nodes in case of elastic scaling. As PatchIndexes are coupled with their partitions, they are also reassigned automatically and can generally be restored from disk or the log file on the reassigned node.

The PatchIndex is currently designed as an in-memory data structure. The index creation is logged to a write-ahead log (WAL), so the index can be reconstructed when performing the log replay in case of a system restart, failure or scaling. The determined patches are not written to the WAL in order to keep it slim, so the index is reconstructed from the data using the same mechanisms as for index creation. There are several alternatives to the in-memory approach that should be evaluated in the future. First, the index data could be materialized to disk, which has the advantages of durability, easy recovery and reducing the main memory consumption, as not all PatchIndexes have to be held in-memory at the same time. On the contrary, reading the relevant PatchIndex data from disk during query execution might decrease query performance, harming the desired benefit from the PatchIndex usage. Second, the PatchIndex information could be materialized as a bitmap column to the table and scanned by ordinary table scan operators.

6 Parallel query processing

In this section, we present approaches to integrate PatchIndexes into parallel and distributed query processing. We present parallel index creation and index scan concepts as well as use cases to integrate the PatchIndex into parallel query execution. We assume the underlying database system to run in a parallel single-server environment or a distributed cluster, which might consist of nodes of arbitrary, commodity hardware or virtual machines in the cloud environment. These cluster nodes are assumed to be connected over network and are based on a shared-nothing architecture [24]. Partitioned data as described in Sect. 5 has to be accessible by e.g. being stored in a cloud file system.

6.1 Index creation

After executing the index-definition-query to create a PatchIndex, we invoke the initial index filling as a postquery. Here we need to determine the set of patches P_c for each partition-local index, which varies for the different constraints. For NUC, the main challenge is the fact that uniqueness is a global constraint. In a general case, the occurence of non-unique values might be scattered among different partitions, resulting in the need for communication for their discovery. In order to determine the set of patches P_c for NUC, we utilize the parallel query plan in Fig. 3 for the discovery query stated in Sect. 4. The inner query, namely discovering all non-unique values, is shown in the right part of the query plan. In the general case, we need to repartition the data on the indexed column c to enable the grouping on c in the next step. Afterwards, we filter for groups with more than one tuple and build a shared hash table on the results. The creation of the shared hash table involves

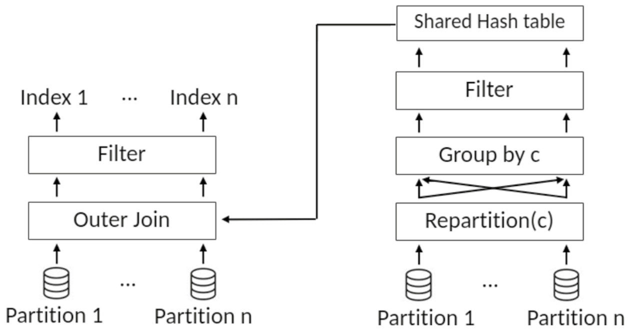

Fig. 3 Parallel PatchIndex creation for uniqueness constraint

communication between the partitions but the hash table is then shared between all partitions. As a consequence, the outer join and the subsequent filter for determining the tuple identifiers of all occurences of non-unique values can then be executed in parallel without any communication and the results can be directly appended to the respective partitions. Note that partitions are cached in memory as much as possible, so reading a partition twice is avoided in the query. In conclusion, we have two communication points in this query, which is the repartitioning step and the creation of the shared hash table. In cases where data is already partitioned on the indexed column *c*, the repartitioning as well as building a shared hash table becomes obsolete. Hash tables can be built partition-locally, resulting in a parallel execution without communication for this specific case.

The creation process for NSC is much more simple. The sorting is determined partition-locally and we integrated the computation of the longest sorted subsequence into the PatchIndex append operator. This way, no communication is needed and the creation query can be fully run in parallel.

6.2 Index scan

In order to apply the PatchIndex information to the dataflow during query execution, we designed the PatchIndex scan by combining an ordinary partitioned table scan with specialized partitioned selection operators with modes *exclude_patches* and *use_patches*. The goal of the PatchIndex scan is to split the dataflow of the scan operator into two distinct dataflows, one containing tuples satisfying a certain constraint and one containing exceptions to the constraint. Referring to Definition 2 of Sect. 3, selection mode *exclude_patches* produces $R_{\setminus P}$, while mode *use_patches* returns R_P.

Once during the build phase of the query, these specialized selection operators query the PatchIndex that belongs to the scanned partition to receive a pointer to the list of patches for the identifier-based approach or a pointer to the bitmap for the bitmap-based approach. The pointer as well as other metadata like processed tuples are stored in a state variable of the operator. During query execution, both modes pass the incoming dataflow to the next operator while applying the patch information on-the-fly using a merge strategy for the identifier-based approach. Therefore we use the elements of the set of patches P_c in a sorted way (Note that the both discovery methods automatically produce the order. Otherwise the elements would need to be sorted during index creation). The concept of the merge strategy for *exclude_patches* is shown in Algorithm 1 and is based on maintaining a patch pointer to the next element in the patch array and increasing the pointer once the element is applied.

For *exclude_patches*, applying patch information means skipping a matching tuple and is realized in lines 11 to 15. If the condition in line 11 is not satisfied, the commented condition in line 13 is ensured as the set of patches P_c is sorted and as the patch pointer is increased by one for each match. The mode *use_patches* only passes elements that match elements of the patch array. For this, the conditions in lines 11 and 13 of Algorithm 1 are exchanged and the patch pointer is increased before returning the tuple, also making the else branch obsolete. Additionally, we return NULL in the case that all patches are already processed in line 7. For the bitmap-based approach, both selection modes are realized using a lookup operation on the bitmap holding the patch information. If a bit in the bitmap is set, the respective tuple passes the *use_patches* mode, while passing the *exclude_patches* mode otherwise.

Algorithm 1 Identifier-based ExcludePatches.Next

```
Input: SelectionState state
 1: while TRUE do
 2:     tuple ← scan.next()
 3:     if tuple == NULL then
 4:         return NULL
 5:     end if
 6:     if state.patch_pointer ≥ state.num_patches then
 7:         return tuple
 8:     end if
 9:     next_patch_id ← state.patches[state.patch_pointer]
10:     state.processed_tuples++
11:     if state.processed_tuples < next_patch_id then
12:         return tuple
13:     else // state.processed_tuples == next_patch_id
14:         state.patch_pointer++
15:     end if
16: end while
```

In order to reduce I/O effort of scan operators, data pruning by applying scan ranges to scan operators is a common concept of analytical database systems. These scan ranges are often computed by small materialized aggregates [18], which are also supported by the PatchIndex scan. While building the query plan, the specialized selection operators with modes *exclude_patches* and *use_patches* fetch the scan ranges from the scan operators. During query execution, they merge the scan ranges on-the-fly with the patches by adjusting the *patch pointer* in order to skip patches outside the ranges or computing an offset within the bitmap. Applying scan ranges to scan operators decreases the number of scanned tuples. As we computed the set of patches P_c on the full set of values for the indexed column c, the selected patches are not accurate anymore. For the case of NSC, pruning tuples from the table may result in the sorted subsequence not being the longest sorted subsequence anymore. Nevertheless, pruning tuples from a sorted subsequence keeps the sequence sorted. For NUC, values that were unique within the full table stay unique for the pruned table as well. As a consequence, merging scan ranges with patches does not harm the correctness of the query result.

6.3 Analytical queries

The key for efficient query processing in parallel and distributed databases is to execute queries locally on partitions as long as possible and ideally just combine intermediate results before returning them. With our approach to support partitioning by creating separate PatchIndexes for partitions we follow this execution paradigm and enable systems to integrate PatchIndexes into efficient parallel and distributed query processing. Here we benefit from putting a larger one-time effort into index creation like described in Sect. 6.1. In the following, we present three use cases of integrating PatchIndexes into query execution, namely distinct, sort and join operators. The general goal of the query optimizations using PatchIndexes is to drop these expensive operations on data that is known to satisfy a constraint and achieve a potential query speedup.

6.3.1 Distinct operator

The information about NUC can be exploited in distinct queries on an indexed column *c*. Here the most expensive operator is the distinct operator, which is typically realized using a hash-based or sort-based aggregation. A potential query plan is shown in the left part of Fig. 4. After splitting the dataflow using the PatchIndex scan, we only have to compute the distinct aggregation on the dataflow containing the exceptions to the uniqueness constraint, which might be the minor part of the data. Values of the second dataflow are ensured to be unique after excluding exceptions. Afterwards, both dataflows are combined using a partition-local union operator. The query is hereby allowed to contain additional pre-selections or projections

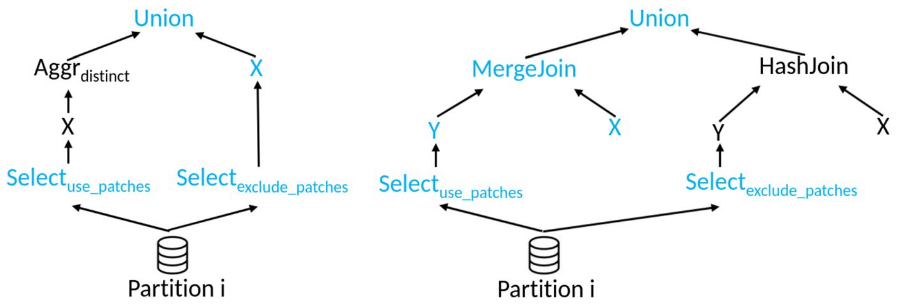

Fig. 4 Query plans for distinct (left) and join (right) queries after PatchIndex optimization with the newly inserted operators highlighted

before the aggregation (abstracted as "X" in the query tree), which are also copied to both subtrees. This query plan can be run locally for each separate partition. In cases where data is not partitioned on the aggregation column, the aggregation requires a repartitioning beforehand, which is also copied to both subtrees. If data is not required to be partitioned after the union operator, e.g., if there are no subsequent operations, the repartitioning is dropped from the dataflow containing the unique values.

6.3.2 Sort operator

The information about NSC can be exploited in sort queries that require a sort operator on a column *c* a PatchIndex is defined on. Similar to the query plan of distinct queries, we can here drop the sort operator on the dataflow that excluded the exceptions, as these values are known to be sorted. The query plan is similar to the left part of Fig. 4, exchanging the aggregation operator with the sort operator. Additionally, both dataflows need to be combined using a partition-local merge operation instead of a union operator to preserve the sort order. As a result, this query can be executed partition-locally without the need for any communication.

6.3.3 Join operator

As a second use case, NSCs can also be integrated into partitioned join queries as shown in the right part of Fig. 4. Here we can replace the partitioned HashJoin operator with the more efficient MergeJoin for the sorted subsequence. As a requirement, both join sides have to be partitioned on the join attribute and the other join side (abstracted as "X" in the query tree) needs to be sorted on the join attribute. This is a common case for joins between fact tables and dimension tables in data warehouses, where dimension tables are typically partitioned and sorted on their dimension key. Pre-selections and projections are also allowed before the join operation (abstracted as "Y" in the query tree), and it is also allowed to contain join operators that do not change the partitioning, so e.g. being the probe side of a HashJoin. While the sorted subsequence is joined using the MergeJoin, the exceptions are joined using an ordinary HashJoin and both dataflows are combined using a partition-local union operator. As the number of exceptions is known during query optimization, the join sides of the HashJoin can be chosen optimally by building the hash table on the side with smaller cardinality. Additionally, the query subtree "X" is cached to avoid unnecessary computations.

7 Index selection

Choosing columns to create PatchIndexes differs from the classic index selection problem in database systems [4]. As PatchIndexes are designed to not change the way data is organized physically, multiple PatchIndexes can be defined on single tables or even single columns. Therefore, choosing columns for PatchIndex creation can be based on a local decision whether a column is suitable or not.

Surely, this can be decided by a database administrator who is familiar with the database schema and the included data. As this requirement is not satisfied in many use cases, especially for cloud applications, we provide PatchIndex selection approaches for self-managing systems in the following.

In a workload-adaptive PatchIndex selection strategy, we can use information and statistics from typical workload queries to base the decisions on. This way, the database schema is redefined and adapted to the workload over time by the creation of PatchIndexes. The key of this approach is the integration of leightweight statistic functions into query operators that can be accelerated using PatchIndexes. For NUC, the most important information is the selectivity s of distinct aggregations, which can be defined as the ratio of output tuples to input tuples. A distinct aggregation with a high selectivity (a ratio near 1) produces many output values from the given input values, which means that many of the input values were unique. Therefore, columns with high selectivities are reasonable candidates for the creation of a PatchIndex. We additionally consider the ratio t of input tuples to the cardinality of the whole table, as the selectivity of the aggregation operator might be influenced by preceeding filter operations. Both ratios are then linearly combined to determine a score for a given column being a PatchIndex candidate:

$$score = p \cdot s + (1 - p) \cdot t \quad with \quad score, p \in [0, 1] \tag{1}$$

As a result, a high score indicates that a large amount of tuples were unique in a large subset of the data, which indicates that the examined column is a good candidate for PatchIndex creation.

In a workload-agnostic PatchIndex selection strategy, we actually test all columns for the given constraints. For NUC, we count the occurences of non-unique values using a query similar to the creation query shown in Fig. 3. For NSC, we utilize the longest sorted subsequence algorithm that is integrated in the PatchIndex creation algorithm. In order to accelerate this testing and potentially prune bad candidates early, we use sampling of the column values. With e being the ration of exceptions to input tuples, the selectivity $s = 1 - e$ and t being the sample fraction, we can utilize the score calculation from Eq. 1 in this approach to rank PatchIndex candidates. Note that sampling might produce wrong assumptions if samples are chosen badly. For example, it might occur that all values in the samples are unique, but no values are unique in the actual column, or that no values in the sample are unique and all remaining values would be unique. Therefore, samples should be chosen randomly and the sample size should be a significant part of the actual data.

For both strategies we can further limit the amount of memory that can be used by PatchIndexes. As the memory consumption is known before creation (constant for bitmap-based approach or linearly in the number of exceptions for the identifier based approach), we can order columns by their score and select the best columns until the specified memory is exhausted. This problem is similar to the knapsack problem and we use a greedy algorithm for the decision, which potentially chooses a good but non-optimal solution but is more efficient than exact solutions using dynamic programming.

8 Evaluation

In our evaluation we show the impact of PatchIndexes on query performance as well as the index creation effort and memory consumption. We integrated PatchIndexes into the Actian Vector 6.0 commercial DBMS, which is built on the X100/Vectorwise [2] analytical database engine. X100 relies on partitioning as the key for parallel query processing. The system runs on a machine consisting of two Intel(R) Xeon(R) CPU E5-2680 v3 with 2.50 GHz, offering 12 physical cores each, 256 GB DDR4 RAM and 12 TB SSD. For all measured results, we used queries on hot data, which means that data resides in the in-memory buffers of the system. This way, we reduce the I/O impact and focus on the pure query execution time.

In order to evaluate the impact of the exception rate in the test data on the PatchIndex performance, we designed a data generator [11] that varies exception rates for both constraints. The data consists of 1B tuples with two columns, a unique *key* column and a *value* column that shows the desired data distribution. In order to exploit parallel data processing, we partition the datasets on the *key* column into 24 partitions when loading it into the system. As the *key* column is unique, this results in partitions of nearly equal size. For the uniqueness constraint, the exceptions of the *value* column are equally distributed into 100 K values, while the remaining values are unique and differ from the values of the exceptions. For the sorting constraint, exceptions are randomly chosen and all remaining values form a sorted sequence in ascending order. For both constraints, exceptions are randomly placed in the datasets. As the datasets are generated once before the benchmarks, the randomness does not impact the comparability of the evaluation results.

We compared the generic PatchIndex structure against different specialized materialization approaches for the respective constraints that a user would use to accelerate queries. For distinct queries, we used materialized views as a comparison, which is a widely used technique in database systems to pre-compute partial queries like the distinct query in our example. This way, expensive distinct queries are replaced by simple scan queries on the materialized view. For sort queries, we compared PatchIndexes against SortKeys, which physically sorts data on the given SortKey column. This way, sort queries can be translated to simple scan queries. Last, we evaluated join queries by comparing the PatchIndex approach against JoinIndexes [25], which materialize foreign key joins as an additional table column. If a SortKey is defined on the table holding the primary key of the join, the foreign key related table is ordered similarly, so that a MergeJoin becomes possible to join both tables.

8.1 Query performance

In order to examine query performance, we ran a distinct query for NUC and a sort query for NSC on the *value* column on the test data. Figure 5 shows the results of the experiments. For NUC, reference runtimes without any constraint definition increase with increasing exception rates, before decreasing starting from an exception rate of 0.3. With increasing exception rates, the number of distinct tuples and therefore the

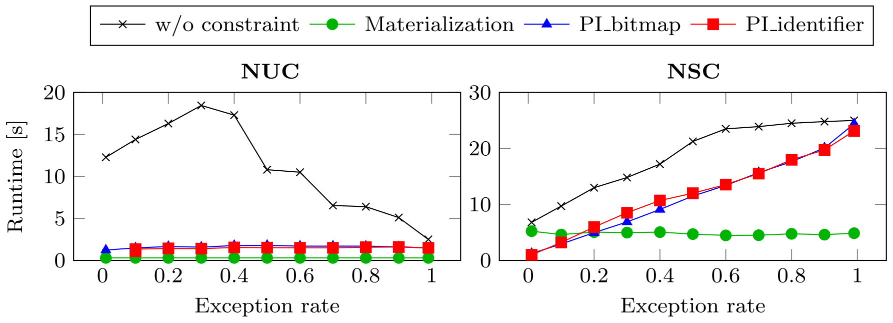

Fig. 5 Runtimes of a distinct/sort query with varying exception rate

number of aggregation groups decrease. The runtime behaviour is then caused by the inference of a reduced hash table size on the one hand and the increased communication cost on the other hand, as the system uses a shared hash table build plan for the aggregation. Using a PatchIndex leads to a significant performance impact comparable to the materialized view for the distinct query also for very high exception rates, with both design approaches performing similarly in this experiment. The runtime of the PatchIndex supported query slightly increases with increasing exception rates due to more tuples being processed in the aggregation. Nevertheless, using a PatchIndex does not show a negative performance impact for this experiment.

For NSC, reference runtime increases with increasing exception rates. This is a result of the internal quicksort pivoting strategy, which behaves better the more sorted the input already is. Using a PatchIndex again shows a significant performance speedup that shrinks expectedly with increasing exception rates, as more tuples need to be processed in the sort operator. Again, using a PatchIndex does not impact performance in a negative way for this case. Using a SortKey shows a constant runtime slightly higher than the scan of the materialized view, as partitions need to be merged here to ensure the tuple order. Additionally, Vector performs a Sort operator to ensure the sorting constraint, leading to slightly worse performance than using PatchIndexes for small exception rates.

8.2 Creation runtime

Besides query performance, the effort to create a PatchIndex is an important measure for the usability of the index structure. Similarly to other index structures, the PatchIndex structure is designed to be used multiple times, so investing a one-time effort is worth the performance improvement for multiple subsequent queries. Figure 6 shows the creation runtimes for NUC and NSC with varying exception rates. For NUC, the runtime basically follows the reference runtime of the distinct query in Fig. 5, as this query is pre-computed here. Comparing the PatchIndex creation to the materialized view creation, storing the information in the index structure leads to a small overhead in runtime. As explained in Sect. 6.1, creating PatchIndexes for NUCs requires communication for

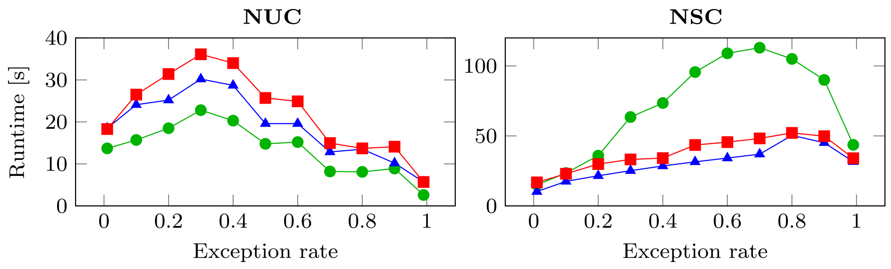

Fig. 6 Runtime for materialization/index creation for varying exception rate

repartitioning. In our experiments, the communication overhead was constantly responsible for around 55–65% of the creation runtime for reshuffling the constant amount of data. For NSC, increasing exception rates lead to more comparisons in the longest sorted subsequence algorithm, but also decreases the length of the sorted sequence and therefore decreases the effort to reconstruct it. The inference of these parts leads to the observed behaviour. For both constraints, the bitmap-based approach performs slightly better in this experiment, as setting bits in a pre-allocated bitmap performs better than maintaining a growing list of identifiers. In comparison, creating a Sortkey takes significantly more time due to the physical reorganization of the table.

8.3 Memory consumption

The memory consumption of a PatchIndex is independent from the materialized constraint it holds and shown in Table 1 for the example dataset and example exception rates e. While the bitmap-based approach has a constant total memory consumption (1 bit per tuple), the memory consumption of the identifier-based approach grows linearly the number of exceptions. As described in Section 5, the bitmap-based approach has a lower memory consumption for all cases with exception rate $e > 0.0156$. In comparison, storing a materialized view is significantly more expensive and depending on the number of unique values. Materializing a JoinIndex requires an additional column of 64 Bit values. For comparability, the memory consumption is compared without applying compression.

Table 1 Memory consumption for example dataset of $t = 10^9$ tuples (without compression)

	PI_bitmap	PI_identifier	Mat. view (NUC)	JoinIndex
General	$t/8 \cdot 1.0039$B	$e \cdot t \cdot 8$B	$(10^5 + (1 - e) \cdot t) \cdot 8$ B	$t \cdot 8$ B
$e = 0.01$	125.48MB	80MB	7.9GB	64GB
$e = 0.2$	125.48MB	1.6GB	6.4GB	64GB

8.4 TPC-H

For the evaluation of join queries, we used the popular and well-known TPC-H benchmark [3]. Although it only contains clean data with perfect constraints, PatchIndexes can also be used in this environment to provide a comparable evaluation. In our experiments, we used the benchmark at scale factor SF 1000 and focused on the largest join between the tables lineitem and orders, resulting in a subset of evaluated queries that contain this join. The reference run consists of the foreign key join without further indexes. For the PatchIndex run we only used the bitmap-based design approach, which showed a significantly better memory footprint in the memory experiments. As the benchmark only contains perfect constraints, queries are further optimized using zero-branch-pruning (ZBP), dropping query subtrees that are ensured to not produce any tuples. This way, the query subtrees that handle the exceptions are pruned from the query plan, resulting in better performance. As the spezialized materialization approach, we built a JoinIndex on the foreign key and co-sorted both tables to enable an efficient merge join.

Figure 7 shows the measured query runtimes. For Q3 and Q7 we can observe a major performance benefit of 33% and 25% respectively when using PatchIndexes over the reference runtime. Additionally activating zero-branch-pruning reduces the overhead introduced by the Patchindex optimization, leading the queries to run 43% and 40% faster compared to the reference runtime. For both queries, using PatchIndexes achieves a query performance similar to the JoinIndex variant and also better when using zero-branch-pruning, which is caused by the additional effort to scan the JoinIndex column. In Q12 we can observe a different behaviour. As the lineitem-orders join is quite small in this query due to preceeding filters, the overhead of splitting the query tree to make use of the PatchIndex information is larger than it's benefit, resulting in a slightly worse query performance. Therefore, the optimizer would not have chosen this query plan. However, using ZBP results in a better performance and a performance gain compared to the reference runtime. Regarding creation effort, creating a PatchIndex took around 100 s for this use case, while creating a JoinIndex took significantly more time of about 600 s.

Although the benchmark only contains perfect constraints, it is an example for another advantage of the PatchIndex approach. Even if a dataset is clean at a point in time, it may become unclean in the future by update operations. While these updates would be aborted with the definition of usual constraints, PatchIndexes would allow

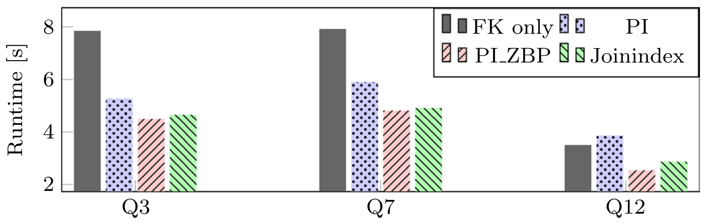

Fig. 7 TPC-H query performance for SF1000

the updates and the respective transition from a perfect constraint to an approximate constraint.

8.5 Resume

Our evaluation showed that PatchIndexes can significantly improve query performance for distinct, sort and join queries even for high exception rates and has a performance impact comparable to the specialized materialization techniques of materialized views, SortKeys and JoinIndexes. The bitmap-based design approach showed a better memory consumption for these cases and a slightly better index creation runtime compared to the identifier-based approach.

9 Further use cases

In the previous sections we presented approaches to exploit “nearly unique columns” and “nearly sorted columns” in parallel and distributed query execution. As we designed our index structure in a generic way, the PatchIndex is not limited to these two constraints, but can be used for many other use cases. In order to adapt the concept to other use cases, a developer would only need to specify an index creation approach and query optimization rules to integrate the index into query execution and actually benefit from the index definition. In the following, we present potential additional use cases for PatchIndexes.

9.1 Other formal constraints

Formal database constraints are not limited to uniqueness and sorting constraints. One could apply the PatchIndex approach also for, e.g., “constness” of column values, potentially dropping filters on these columns from query plans.

9.2 User-defined constraints

Constraints are not necessarily formal database constraints, but could also be user-specific, semantic constraints. Examples for user-defined constraints are various, e.g. nearly all dates are later than a given year, nearly all people come from the same country or nearly all monetary values are between a certain range. Filtering on these attributes can then be accelerated with PatchIndexes by pruning data and reducing scan effort.

9.3 Approximate functional dependencies

Besides approximate constraints, approximate functional dependencies are also covered in research [13]. PatchIndexes could also be applied here and integrated into integrity checks or cardinality estimation.

9.4 Approximate foreign keys

Foreign keys require that every foreign key matches a primary key of the related table and joining tables without foreign keys leads to a N-to-M join. Allowing exceptions to this requirement using PatchIndexes could be exploited in query execution by using a more efficient 1-to-N join for all tuples satisfying the foreign key relationship and only relying on the general N-to-M join for exceptions. Additionally, approximate foreign keys could be used in join cardinality estimation, as tuples satisfying the constraint match with exactly one join partner.

9.5 Approximate query processing

PatchIndexes could be applied to the field of approximate query processing [15]. As an example, approximate count distinct queries could be efficiently answered by simply querying a PatchIndex for a NUC without the need to execute an aggregation.

10 Conclusion

In this paper, we motivated the problem of exploiting approximate constraints in query execution. We designed a generic PatchIndex data structure that maintains exceptions to arbitrary constraints, for which we provided a dense and a sparse design approach. As examples for approximate constraints, we discussed "nearly unique columns" (NUC) and "nearly sorted columns", for which we described discovery approaches and use cases for query execution. Adapting the concept of PatchIndexes to these examples, we showed how to efficiently create PatchIndexes and integrate them into query execution using the PatchIndex scan. Additionally, we provided query optimization techniques for distinct, sort and join queries using NUCs and NSCs. We hereby applied our approaches on the parallel and distributed database environment, where data is usually partitioned and distributed. Our evaluation showed that using PatchIndexes can significantly increase query performance. As the definition of approximate constraints is not possible in ordinary database systems, PatchIndexes can also improve schema quality and prevent the loss of useful information on the data. Furthermore we provided heuristics to automatically discover candidate columns for PatchIndex creation, which can be integrated into arbitrary database self-managing tools.

As the idea of PatchIndexes is able to improve query performance, we plan to further enhance the concept. The feature of maintaining exceptions of constraints offer opportunities for lightweight support for table inserts, deletes and updates. We especially aim at enabling update operations to global constraints (like the uniqueness constraint) while avoiding a full table scan, potentially outperforming spezialized materialization approaches like materialized views, SortKeys and JoinKeys. Additionally, alternatives to the in-memory design should be evaluated as well as the described alternative use cases for PatchIndexes.

Acknowledgements This work was funded by the Actian Corp.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: a survey. VLDB J. **24**(4), 557–581 (2015). https://doi.org/10.1007/s00778-015-0389-y
2. Boncz, P.A., Zukowski, M., Nes, N.: MonetDB/X100: hyper-pipelining query execution. In: CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, pp. 225–237 (2005), http://cidrdb.org/cidr2005/papers/P19.pdf
3. Boncz, P.A., Neumann, T., Erling, O.: TPC-H analyzed: hidden messages and lessons learned from an influential benchmark. In: Performance Characterization and Benchmarking, vol 8391, Lecture Notes in Computer Science, Springer, Cham, pp. 61–76, https://doi.org/10.1007/978-3-319-04936-6_5 (2014)
4. Comer, D.: The difficulty of optimum index selection. ACM Trans. Datab. Syst. **3**(4), 440–445 (1978). https://doi.org/10.1145/320289.320296
5. Fredman, M.L.: On computing the length of longest increasing subsequences. Discret. Math. **11**(1), 29–35 (1975). https://doi.org/10.1016/0012-365X(75)90103-X
6. Ghita, B., Tomé, D.G., Boncz, P.A.: White-box compression: learning and exploiting compact table representations. In: CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam. http://cidrdb.org/cidr2020/papers/p4-ghita-cidr20.pdf (2020)
7. Graefe, G.: Modern B-tree techniques. Found. Trends Databases **3**(4):203–402 (2011), https://doi.org/10.1561/1900000028
8. Gunopulos, D., Khardon, R., Mannila, H., Saluja, S., Toivonen, H., Sharma, R.S.: Discovering all most specific sentences. ACM Trans. Database Syst. **28**(2), 140–174 (2003). https://doi.org/10.1145/777943.777945
9. Heise, A., Quiané-Ruiz, J.A., Abedjan, Z., Jentzsch, A., Naumann, F.: Scalable discovery of unique column combinations. Proc. VLDB Endow. **7**(4), 301–312 (2013). https://doi.org/10.14778/2732240.2732248
10. Huhtala, Y.: Tane: an efficient algorithm for discovering functional and approximate dependencies. Comput. J. **42**(2), 100–111 (1999). https://doi.org/10.1093/comjnl/42.2.100
11. Kläbe, S.: Data Generator (2020). https://github.com/Sklaebe/Approximate-Constraint-Data-Generator
12. Kläbe, S., Sattler, K.U., Baumann, S.: PatchIndex: exploiting approximate constraints in self-managing databases. In: 2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW), pp 139–146 (2020), https://doi.org/10.1109/ICDEW49219.2020.00014, iSSN: 2473-3490
13. Kruse, S., Naumann, F.: Efficient discovery of approximate dependencies. Proc. VLDB Endow. **11**(7), 759–772 (2018). https://doi.org/10.14778/3192965.3192968
14. Köhler, H., Link, S., Zhou, X.: Possible and certain SQL keys. Proc. VLDB Endow. **8**(11), 1118–1129 (2015). https://doi.org/10.14778/2809974.2809975
15. Li, K., Li, G.: Approximate query processing: what is new and where to go? Data Sci. Eng. **3**(4), 379–397 (2018). https://doi.org/10.1007/s41019-018-0074-4
16. Livshits, E., Heidari, A., Ilyas, I.F., Kimelfeld, B.: Approximate denial constraints. Proc. VLDB Endow. **13**(10), 1682–1695 (2020). https://doi.org/10.14778/3401960.3401966

17. Mitzenmacher, M.: Compressed bloom filters. IEEE/ACM Trans. Netw. **10**(5), 604–612 (2002). https://doi.org/10.1109/TNET.2002.803864
18. Moerkotte, G.: Small materialized aggregates: a light weight index structure for data warehousing. In: Proceedings of the 24rd International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., San Francisco, VLDB '98, pp 476–487 (1998), http://dl.acm.org/citation.cfm?id=645924.671173
19. Papenbrock, T., Naumann, F.: A hybrid approach for efficient unique column combination discovery. In: Mitschang, B., Nicklas, D., Leymann, F., Schöning, H., Herschel, M., Teubner, J., Härder, T., Kopp, O., Wieland, M. (eds.) Datenbanksysteme für Business, Technologie und Web (BTW 2017), pp. 195–204. Gesellschaft für Informatik, Bonn (2017)
20. Pena, E.H.M., de Almeida, E.C., Naumann, F.: Discovery of approximate (and exact) denial constraints. Proc. VLDB Endow. **13**(3), 266–278 (2019). https://doi.org/10.14778/3368289.3368293
21. Rahm, E., Do, H.: Data Cleaning: problems and current approaches. IEEE Data Eng. Bull. **23**, 3–13 (2000)
22. Rostin, A., Albrecht, O., Bauckmann, J., Naumann, F., Leser, U.: A machine learning approach to foreign key discovery. In: 12th International Workshop on the Web and Databases, WebDB 2009, Providence, Rhode Island, (2009) http://webdb09.cse.buffalo.edu/papers/Paper30/rostin_et_al_final.pdf
23. Saxena, H., Golab, L., Ilyas, I.F.: Distributed implementations of dependency discovery algorithms. Proc. VLDB Endow. **12**(11), 1624–1636 (2019)
24. Stonebraker, M.: The case for shared nothing. IEEE Database Eng. Bull. **9**, 4–9 (1985)
25. Valduriez, P.: Join indices. ACM Trans. Database Syst. **12**(2), 218–246 (1987). https://doi.org/10.1145/22952.22955
26. Vogelsgesang, A., Haubenschild, M., Finis, J., Kemper, A., Leis, V., Muehlbauer, T., Neumann, T., Then, M.: Get Real: how benchmarks fail to represent the real world. In: Proceedings of the Workshop on Testing Database Systems, Association for Computing Machinery, Houston, DBTest'18, pp. 1–6 (2018), https://doi.org/10.1145/3209950.3209952
27. Wei, Z., Leck, U., Link, S.: Discovery and ranking of embedded uniqueness constraints. PVLDB **12**(13), 2339–2352 (2019)
28. Zukowski, M., Héman, S., Nes, N., Boncz, P.A.: Super-scalar RAM-CPU cache compression. In: Liu L, Reuter A, Whang KY, Zhang J (eds) Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, Atlanta, IEEE Computer Society, p. 59 (2006), https://doi.org/10.1109/ICDE.2006.150