PatchIndex - Exploiting Approximate Constraints in Self-managing Databases

Steffen Kläbe
TU Ilmenau, Germany
steffen.klaebe@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

Stephan Baumann
Actian Germany GmbH
stephan.baumann@actian.com

***Abstract*—In the cloud environment, data warehouse solutions need to be self-managing in order to be usable without prior database administration knowledge. Additionally, data is typically not clean in these environments, as it is imported from various sources. As a consequence, automatic schema optimization as an important task of self-management becomes difficult without human interaction and data cleaning steps. Within this paper, we focus on constraint discovery as a subtask of schema optimization. Real world datasets with unclean data may not contain perfect constraints, as a minor part of the values hampers the definition of them. Therefore, we introduce the PatchIndex structure, which handles these exceptions to column constraints and enables self-management tools to discover and define approximate constraints on unclean data. We present "nearly unique column" and "nearly sorted column" constraints, both managed by the generic PatchIndex structure. Furthermore, we provide mechanisms to discover these constraints and show how query performance can benefit from them for different use cases by integrating them into query optimization. Our evaluation shows that the PatchIndex structure offers opportunities for a significant performance boost in different use cases while enabling self-management tools to define constraints on unclean data.**

***Index Terms*—self-managing databases, schema refinement, approximate constraints, uniqueness, patch processing**

I. Introduction

Bringing database management systems (DBMS) to the cloud environment lowers the barrier to start data analysis by making DBMS accessible within minutes and avoiding administration effort of on-premise solutions. As a consequence, users are enabled to start working with their data and achieve business value out of it without the requirement of prior database administration knowledge. Therefore, applications using cloud databases typically lack database administrators and raise the need for self-managing databases. Tools for automatic database administration cover a wide range of tasks, e.g., schema refinement, index selection, constraint discovery or data cleaning.

With this paper, we want to address the problem of automatic constraint discovery. In big data environments, datasets are typically unclean. Therefore, it becomes challenging, if not even impossible, for automatic tools to discover perfect constraints like primary keys, uniqueness constraints or sorting constraints without human interaction to decide for data cleaning steps. This results in database schemes with sparse schema information. As constraints are used in query optimization, e.g., selecting the actual join algorithm or estimating cardinalities of intermediate results, non-optimal query plans and non-optimal query performance are a consequence. Additionally, even if constraints are defined, maintaining and enforcing them is expensive and therefore often avoided by users.

Except potentially non-existing perfect constraints, these datasets may include approximate constraints, which are constraints that hold for nearly all tuples except a number of exceptions. These constraints occur for different reasons in big data environments, e.g.:

- **Data integration:** Data is integrated from various sources. This may lead to duplicates or an arbitrary order based on the order of integration.
- **Missing values:** Integrating different data schemas into one common schema may lead to NULL values.
- **Real world anomalies:** Real world datasets may not contain perfect constraints, e.g. equal names of persons, shared telephone numbers or shared addresses.

Approximate constraints contain valuable information that cannot be used for query execution, as the constraint definition is prohibited by a small part of the values. In order to prevent these information from being lost and to take advantage of them, allowing the definition and usage of approximate constraint is worth investigating.

Common real world use cases further underline the need for handling approximate constraints. In many cases, users combine analytical database systems with dashboard generation tools to simplify reporting, provide an overview over business data for management processes or utilize the built-in interactive query generators. While definetely increasing the usability of database management systems, these dashboard tools convert user interactions to SQL queries that are very large and complicated in many cases. An example query graph commonly generated by these tools is shown in Figure 1. Here, each subtree is a distinct query on an arbitrary column of the database. Out of the results from these queries, dashboard elements like controllers, drop-down selections or similar are generated. These queries would significantly benefit from any additional information on the processed data, like the described approximate constraints. Furthermore, many real world datasets are timestamp-based, as they are generated by e.g. sensor networks. Not only being sorted on the timestamp, these datasets often show an approximate co-sorting of other columns and the timestamp column, e.g. auto-generated values, version numbers or increasing measurements. These nearly co-sorted columns also occur in sales datasets, e.g increasing order numbers leading to increasing order dates, or increasing order dates leading to increasing ship dates. For these cases, knowing about the approximate sorting of the columns would significantly accelerate sorting queries on these columns.

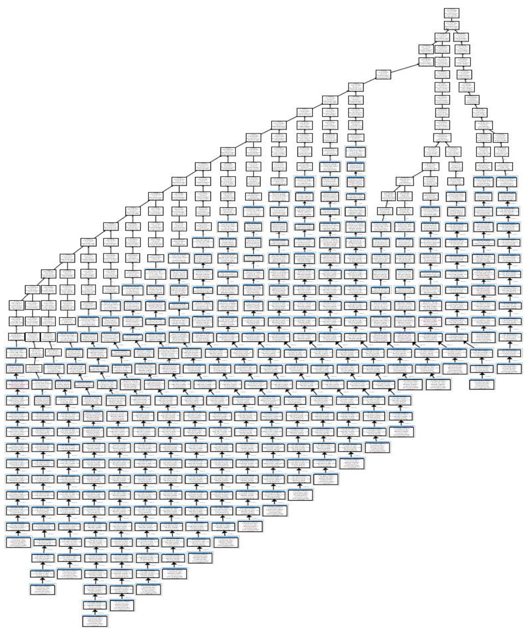
Fig. 1. Overview over a query generated by dashboard tool. Every branch is a distinct query. Graph generated using Actian Vector.

In order to exploit approximate constraints in analytical databases, the main idea of this paper is to handle tuples violating constraints separately to accelerate query execution. We therefore introduce the PatchIndex structure, supporting "nearly sorted columns" and "nearly unique columns" in a common index structure. The main contributions of this paper are:

- We provide methods to discover approximate constraints and to handle them in the PatchIndex structure. These discovery methods can be easily integrated into arbitrary self-managing tools.
- We describe possibilities to benefit from approximate constraints in query planning and query execution and prove the impact in our evaluation.
- By carefully designing the index scan and not changing the way data is physically stored, we enable systems to have multiple (approximate) sort keys within a single table to exploit nearly co-sorted columns.

The remainder of this paper is organized as follows: Section II presents related work in the field of self-management database concepts. In Section III we state basic definitions to base further discussions on. Section IV presents methods to discover the approximate constraints before discussing considerations for the actual PatchIndex design in Section V and describing the integration into a DBMS and query execution in Section VI. We evaluate our approach for different use cases in Section VII before concluding and providing an outlook in Section VIII.

II. Related Work

Constraint discovery is an active research field and based on data profiling techniques [1]. A basic problem of constraint discovery is the discovery of unique column combinations (UCC), which is a set of table columns whose projection only contains unique rows. A UCC is the basic condition for the definition of primary keys and therefore also foreign keys. Finding all exact UCCs for a given relation is shown to be NP-hard [2] and various algorithms like DUCC [3] or HyUCC [4] were introduced to handle the problem complexity by reducing the search space. In [5], not only the search space traversal, but also communication effort is taken into account to cope with the complexity by distributed processing. As candidate classification is a suitable use case for machine learning approaches, candidate pruning is realized using a classificator trained on SQL-extractable features like distinctness, dependencies, value length or column and table names in [6].

Nevertheless all these approaches are limited to exact UCCs, so the occurrence of NULL values remains a problem. Especially for the cloud environment, we can expect that NULL values appear in real world data sets. Possibilities and approaches for data cleaning are numerous and classified in [7]. There are mainly two possibilities to handle NULL values in columns where a constraint is violated by them. Either these tuples are deleted from the data set or missing values are replaced with existing ones by using observed value distributions. While the first one might not be desired, especially if done automatically, the latter one might falsify data analysis. In order to handle NULL values for uniqueness constraints and offering larger possibilities for schema refinement as a consequence, "possible" and "certain" keys were introduced in [8], replacing violating tuple values to enforce constraints. Another approach to handle NULL values was given in [9] with the definition of embedded uniqueness constraints (eUC). The main idea behind eUC is to separate uniqueness from completeness by enforcing uniqueness constraints only on the subset of tuples without the occurrence of NULL values in key columns. The authors give some application examples like schema design and query optimization and provide an implementation approach using existing index structures defined on views without the violating tuples. Furthermore they show that deciding whether a given relation has a eUC is NP-complete and present several discovery algorithms.

Not only taking NULL values into account, the PYRO algorithm is presented in [10] to discover and rank approximate functional dependencies and UCC, extending the pruning rules of the TANE algorithm [11]. The algorithm uses a sampling strategy for candidate pruning to reduce the search space, combined with a separate-and-conquer approach for candidate validation.

The concept of patch processing, handling exceptions to certain distributions or properties of data, is commonly used in the field of compression. The authors of [12] proposed the PFOR, PFOR-DELTA and PDICT compression schemes, making common compression schemes more robust by handling outliers separately. Additionally, the concept of white-box compression was introduced in [13]. Instead of choosing a compression schema for all values of a column, the concept aims at learning functions or properties of the data using an automatic learning approach. Based on this, compression can be optimized by varying compression algorithms between values that follow the observed behaviour and values that are exceptions to this, leading to a significant increase of compression ratios.

III. Definitions

Within this section, we describe the concept of "nearly unique columns" (NUC) and "nearly sorted columns" (NSC). Values of these columns fulfill the uniqueness constraint or the sorting constraint, respectively, except a set of tuples. We collect the rowIDs of tuples that violate the constraint in a set of patches P_c for a column c, so that the respective constraint is fulfilled by all remaining column values of c. An example is shown in Figure 2. The shown integer values of c are unique if we exclude, e.g., tuples with rowIDs in $\{1, 2, 3, 7\}$, while they are sorted if we exclude tuples with rowIDs in $\{4, 7\}$. In the following, we want to provide formal definitions of the outlined concepts to base our further discussions on.

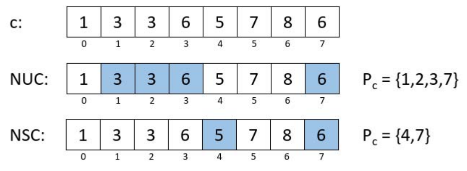

Fig. 2. Example for NUC and NSC for a given dataset

Definition III.1. Naming conventions

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.

Definition III.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 III.3. Threshold variables
We define threshold variables $nuc\_threshold$ and $nsc\_threshold$, both in $[0, 1] \subset \mathbb{R}$.

Definition III.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 want values of c (described using the projection operator $PROJ$ on c) to be unique after we excluded all tuples with tuple identifiers in P_c. The second condition (NUC2) is important to ensure the correctness of the query result, as we later want to utilize the PatchIndex in query execution by querying R_P and $R_{\setminus P}$ separately from each other. In column c of Figure 2, values 3 and 6 are duplicates, so they have to be excluded to make the column unique. As we have to exclude all occurrences of the duplicate values according to (NUC2), P_c consists of four tuple identifiers. Therefore, 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 III.5. Nearly sorted column (NSC)
Given a column c, let $\triangleleft$ 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) $\forall t_i, t_j \in R_{\setminus P} : id(t_i) < id(t_j) \Rightarrow c(t_i) \triangleleft c(t_j)$
(NSC1) $|P_c|/|R| \leq nsc\_threshold$

As an intuition, we want values of c to be sorted according to the given order relation $\triangleleft$ based on the order of their tuple identifiers after we excluded tuples with tuple identifiers in P_c. In order to fulfill the sorting constraint for column c of Figure 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 $nsc\_threshold \geq 0.25$. Nevertheless, this choice is ambiguous, as we can find another set of patches $P_c = \{3, 7\}$ with the same cardinality of two. In the discovery mechanism of NSC presented in Section IV, we are interested in a smallest set P_c.

IV. Constraint discovery

Based on the definitions in Section III, the problem of deciding whether a given column c is a NUC or NSC translates to finding a set of patches P_c such that the respective constraints are fulfilled. Within this section, we give general approaches for both problems that can be integrated into arbitrary automatic database administration tools.

In order to discover a NUC, we need to find all column values that are not unique, which can be realized using a hash table. Instead of realizing this, we can reuse existing operators of database systems, as the described approach is equal to a distinct aggregation. Therefore, we can simply realize the

NUC discovery on SQL level. It is not sufficient to simply compute a distinct query or query all values that occur more than once, as we need all occurences of a non-unique column value in order to meet condition (NUC2). Therefore, we join the result of a aggregation query with the actual table on the examined column c to get the tuple identifiers of all tuples with non-unique values for c, which is equal to the set of patches P_c we want to compute. We need to pay special attention to NULL values, as they are not joined with each other but should be assigned to the set of patches. Therefore, the join operation is realized using an outer join with a subsequent selection, resulting in the following 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
```

By checking for non-null values in *temp* we ensure that we only select values with matching join partners. Based on the result of the query, the classification of a column c as a NUC can be made based on the third condition $|P_c|/|R| \leq nuc\_threshold$.

In order to discover a NSC, we utilize the longest sorted subsequence algorithm [14], for which we need the full data of a given column c as a prerequisite. The algorithm then maintains arrays to keep the length and the predecessor of the last element in the longest sorted subsequence of data $[1, \ldots, k]$ 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). This way we ensure that $R \setminus P_c$ is sorted on column c and as we computed the longest sorted subsequence, we also ensure that the cardinality of P_c is minimal. 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 second condition $|P_c|/|R| \leq nsc\_threshold$.

V. Index design

The PatchIndex data structure is intended to maintain the set of patches P_c for the column c it is defined on. For this, we implemented two basic approaches, which are the identifier-based approach and the bitmap-based approach. For the identifier-based approach, we store the 64 Bit tuple identifiers of all tuples in P_c in an array, which is similar to a sparse approach. As a result, the memory consumption of this approach is proportional to the cardinality of P_c. On the contrary, the bitmap-based approach is similar to a dense way of storing data. With n being the number of tuples in c (or the respective relation R), the PatchIndex holds a bitmap of size n for the column c, which is in particular independent of the cardinality of P_c. The element at position i within this bitmap indicates whether tuple i belongs to P_c or not.

Similar to the decision between a sparse and a dense way of storing, deciding between the identifier-based and the bitmap-based approach is based on the cardinality of P_c. As we require 1 Bit per element for the bitmap-based approach and 64 Bit per element for the identifier-based approach, we can expect that the identifier-based approach has a lower memory consumption for all cases where $|P_c|/|R| \leq 1/64 = 1.56\%$.

In order to fill the PatchIndex structure, we invoke an "AppendToIndex" query as a post-query whenever a PatchIndex is created. The actual computation of P_c varies between the two types of constraints NUC or NSC and is related to the discovery methods described in Section IV. For a NUC on the one hand, the described discovery query is evaluated and the result is passed to the append operator, which simply stores the tuple identifiers or sets the corresponding bit in the PatchIndex. In order to create an index on a NSC on the other hand, a scan of the whole indexed column c is passed to the append operator. The index itself stores the values of c in a temporary array and starts the computation of the longest sorted subsequence once it has the full data set. Based on this sorted subsequence, P_c is computed and stored in the index, before the temporary data array is dropped.

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 or failure. 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. This way, reading the information could be done using ordinary table scan operators.

VI. Integration into DBMS

A. Index Scan

1) Design: In order to benefit from PatchIndexes in query execution, we first have to efficiently apply the patch information to the dataflow of a query execution tree (QET). Therefore, we introduce the PatchedScan. Scan operators typically support the definition of scan ranges to prune data and reduce I/O effort. These scan ranges are typically determined by evaluating selection predicates and using small materialized aggregates [15] that are guaranteed to be filtered out during query optimization. As tuples that harm constraints and are therefore classified as patches might be scattered across the data, translating PatchIndex information into scan ranges would result in numerous fine-grained ranges and non-negligible overhead.

Therefore we decided to realize the PatchedScan by combining an ordinary table scan with a specialized PatchSelect operator placed directly on top of the scan. In order to apply the information of the PatchIndex, we introduce selection modes *use_patches* and *exclude_patches* for these PatchSelect operators. As the PatchIndex works on tuple identifiers, we try to avoid scanning the tuple identifier column by assuming that rowIDs of incoming tuples are equal to tuple identifiers. This is ensured by placing the selection operators directly on top of scan operators, as intermediate operators could harm this assumption by filtering tuples. As a result, the output dataflow of the scan operator is logically splitted by the PatchSelect operators like shown in Figure 3, resulting in a dataflow containing the tuples of the set of patches P_c and a dataflow containing all tuples except the patches. An advantage of this approach is that the physical way tuples of a table are stored is not changed by creating a PatchIndex on that table. As a consequence, it becomes possible to create multiple PatchIndexes on a single table, which significantly differs from the typical limitation of one sort key per table.

Once during the query build phase, selection operators with modes *use_patches* and *exclude_patches* query the PatchIndex in order to receive a pointer to the patch array or to the bitmap holding the patch information. 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 basic 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 16. If the condition in line 11 is not fulfilled, the commented condition in line 14 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 14 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 simply 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 if the respective bit is not set.

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
14:         // state.processed_tuples == next_patch_id
15:         state.patch_pointer++
16:     end if
17: end while
```

2) Partition support: PatchIndexes also support several common techniques of analytical database systems. First, they support partitioning by creating a PatchIndex for each partition separately. This way, partitioning is transparent to the actual index implementation. During query execution, subqueries are executed on partitions as far as possible and we do not harm this ability by placing the specific selection operators directly on top of the scan operators. For NSC, the sorted subsequences are computed for each partition locally, so it is ensured that sorts and MergeJoins can also be evaluated locally (assuming the partitioning attribute of the joined tables match the join attribute). For NUC, the discovery query computes a global grouping in the subquery. As we afterwards join with the partitioned table again, each partition's PatchIndex receives all tuple identifiers for its responsible partition to append.

3) Scan range support: Second, scan ranges are supported by the PatchScan. While building the QET, *exclude_patches* and *use_patches* selection operators fetch the scan ranges from the scan operators below. 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.

B. Query Optimization

We integrated PatchIndexes into query plan rewriting for three different use cases: distinct, sort and join operators.

1) Distinct operator: A use case for NUC is the distinct operator, which is typically realized using a very expensive hash-based aggregation. For this case, we can benefit from the information that the major part of the values of the aggregation column is already unique when a PatchIndex is defined on it. The left part of Figure 3 shows the query plan rewriting for this use case with the highlighted operators being introduced to benefit from the PatchIndex. Starting from a distinct query (shown in black color) consisting of a table scan, followed by an arbitrary subtree X (may consist of selections and non-arithmetic projections) and a distinct aggregation on top, we copy the subtree below the aggregation and insert select operators above the scans of each subtree copy. The select operator with mode *exclude_patches* makes the distinct aggregation unnecessary, as the PatchIndex ensures that values are already unique when excluding the patches. In the second subtree, the select operator with mode *use_patches* only passes the elements of the PatchIndex. We have to apply the distinct operator on these patches, as they are not unique by design of the PatchIndex. As a result, the effort for the expensive aggregation can be avoided for the major part of the tuples. Both dataflows are then combined using a union operator. This use case is currently limited to the lowest aggregation in the query plan, so X is not allowed to contain furter aggregations or join operations.

2) Sort Operator: For NSC, the provided information about a sorted subsequence can be exploited in queries that perform a sort operator on a column c a PatchIndex is defined on. Similar to the rewriting of distinct queries in the left part of Figure 3 (exchanging the aggregation operator with the sort operator), the sort operation only needs to be performed on the set of patches. This significantly reduces the runtime of the sort operator. The dataflow in the right subtree is already sorted on the indexed column c after the patches are excluded by the select operator with mode *exclude_patches*, due to the definition of a NSC. In order to combine both sorted datastreams, the union operator needs to be replaced by a MergeUnion operator to produce a sorted dataflow.

3) Join Operator: The second use case for a NSC is the join operator with the join attribute being the attribute a PatchIndex is built on. Here we can exploit the information that the major part of the table is sorted by using a MergeJoin operator instead of the more expensive HashJoin operator for the sorted subsequence. The right part of Figure 3 shows the rewritten query plan for a join operation between an arbitrary subtree X (needs to be sorted in the join attribute) and a table T with an arbitrary subtree Y above (shown in black color) with operators introduced by this optimization highlighted in blue color. It is currently not allowed that Y contains any joins, so we apply this optimization only on the lowest join. In this case, we can replicate the join subtree and insert the specific select operators above the scans. For the tree that uses the select operator with mode *exclude_patches*, we can exchange the HashJoin with a MergeJoin, as the PatchIndex ensures that the dataflow is sorted when excluding the patches. This way, we can benefit from the faster MergeJoin for the major part of the data. As we know the cardinality of P_c and can estimate the input cardinality for the HashJoin, we can choose the join side with the lower cardinality as the side to build the hash table on as a further improvement.

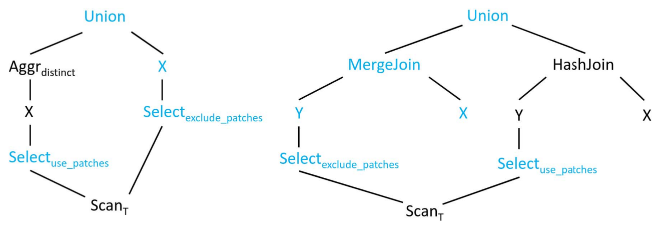

Fig. 3. Distinct query (left) and join query (right) using patches (X, Y are arbitrary subtrees without any join operations)

VII. Evaluation

For our evaluation, we integrated the concept of PatchIndexes into Actian Vector 6.0. The system runs on a machine consisting of two Intel(R) Xeon(R) CPU E5-2680 v3 with 2.50GHz, offering 12 physical cores each, 256 GB DDR4 RAM and 12 TB SSD. As an example workload, we use the TPC-DS [16] benchmark, which is a synthetical decision support benchmark representing a retail database in a snowflake schema. We generated data using the benchmark for scale factor 1000 GB and evaluated two typical use cases for PatchIndexes in this schema. Additionally, we designed a custom data generator for a more fine-grained evaluation of different exception rates in the second experiment. All tables are distributed into 24 partitions.

A. TPC-DS

1) Nearly sorted column: A common use case for NSC is a fact table that is (nearly) sorted on a dimensioning attribute. This especially holds for time-based data, where several columns can be nearly co-sorted according to insertion order of tuples (e.g. order numbers or similar, ship dates co-sorted to order dates). Additionally, dimension tables are typically sorted on their primary key, which is the join key when joining with fact tables. An example in TPC-DS is the catalog_sales.sold_date join with the date dimension. Here we have to exclude 0.5% of the 1.4B tuples to get the sold_date column sorted. This way, we can reduce the total runtime for scanning both relations and joining them from 1.4 seconds to 0.7 seconds.

2) Nearly unique column: For evaluating NUC, we chose columns from the customer table (12M tuples) with different exception rates. Table I shows the measured results in terms of runtime for a count distinct query. While the performance gain is significant for a small amount of exceptions, we can also observe a slight increase in performance even for a very large amount of exceptions.

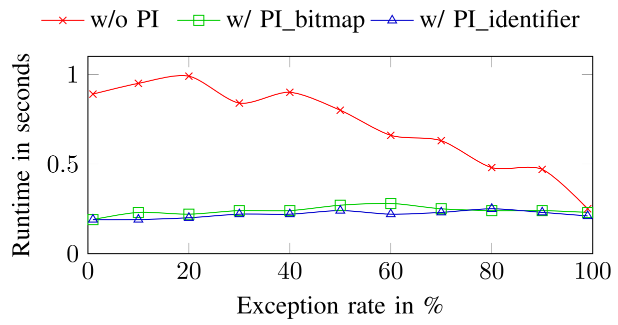

Fig. 4. Runtime for count distinct query with varying exception rate

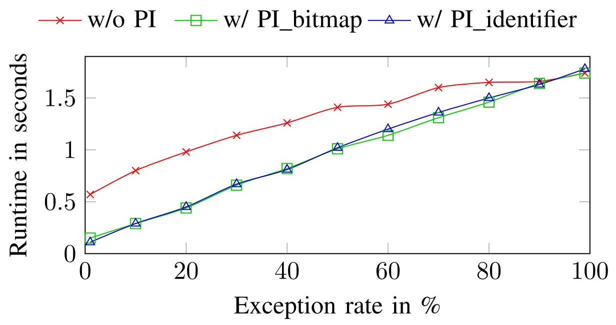

Fig. 5. Runtime for sort query with varying exception rate

Column	Exceptions	w/o PI	w/ PI
c_email_address	3.6%	0.37 s	0.10 s
c_current_addr_sk	86.5%	0.19 s	0.15 s

TABLE I
PERFORMANCE OF NUC PATCHINDEX

B. Varying exception rates

1) Query performance: Using a custom data generator, we generated a dataset of 100M tuples and varied the exceptions for uniqueness and sorting constraints. The exceptions were placed in random locations within the table. After generation, we ran a count distinct query or a sort query respectively on a column with and without a PatchIndex.

For uniqueness, the exceptions were evenly distributed into 100K different values. The results shown in Figure 4 show a significant increase in performance when using the PatchIndex for all exception rates, while both implementation alternatives perform similarly. The runtime when using a PatchIndex slightly increases when increasing the exception rate, which is caused by the increasing number of tuples that have to be evaluated in the distinct aggregation. On the other hand, query runtime when not using a PatchIndex decreases with higher exception rates. Increasing the number of exceptions decreases the number of distinct values, as all exceptions form a fixed number of 100K groups, and therefore also decreases the number of aggregation groups. As a consequence, the distinct aggregation becomes faster the more exceptions to uniqueness the dataset has. Nevertheless, queries using a PatchIndex was faster for all exception rates, as excluding unique values from the actual distinct aggregation significantly decreases the number of aggregation groups to 100K for all exception rates.

For the sorting constraint, we focused on discovering ascending orders. The exceptions in the dataset were generated randomly, so the exception rate varied about $\pm$ 0.1% after discovering the longest subsequence. As the dataset was pre-generated, this variation impacts all measurements for a single exception rate in the same way. The results shown in Figure 5 again show a significant performance gain when using PatchIndexes with both design approaches behaving similarly. Without using a PatchIndex, runtime increases with increasing exception rate. This is caused by the pivoting strategy of the internal QuickSort implementation, behaving better the more sorted the data values already are. When using the PatchIndex, runtime increases with increasing exception rates in a linear way. It shows a higher growth compared to the uniqueness case in Figure 4, caused by the sorting operator not scaling linearly. This way, the actual performance gain shrinks with increasing exception rates, but using a PatchIndex for this use case never leads to worse performance compared to the basic case.

2) PatchIndex creation runtime: Figure 6 shows the runtime for the PatchIndex creation for NUC and NSC varying the exception rate. First, it can be observed that both design approaches behave similarly, as the creation process is dominated by the computation of the exception and the actual insertion of exceptions differs only in a negligible way (inserting an identifier to a list or setting a bit in the bitmap). For NSC, the observed runtimes are a result of the addition of three basic steps: the longest sorted subsequence algorithm, the construction of the exceptions and their insertion. While with increasing exception rates the effort for insertion increases linearly and the effort for construction decreases linearly, the longest sorted subsequence algorithm shows a non-linear behaviour, resulting in the shown runtimes. For NUC, increasing the number of exceptions significantly decreases the number of aggregation groups and accelerates the aggregation as a consequence, like described in Section VII-B1,. As this aggregation dominates the creation process, the overall creation runtime decreases with increasing exception rates. Overall we can state that the PatchIndex creation time is slightly higher than the performance gain it generates in a single usage, so creating a PatchIndex is valuable for use cases that query the indexed column multiple times during the database lifetime, which is the common case.

3) PatchIndex memory consumption: In terms of memory consumption of the PatchIndex data structure, our experiments confirm the expectations we discussed in Section V. For the bitmap-based approach, we measured a constant memory consumption of 12.5 MB for the 100M tuples, while we measured a memory consumption of 7.9 MB per 1% exceptions for the identifier-based approach. As a consequence, the bitmap-based approach should be chosen for use-cases with more than 1.6% exceptions.

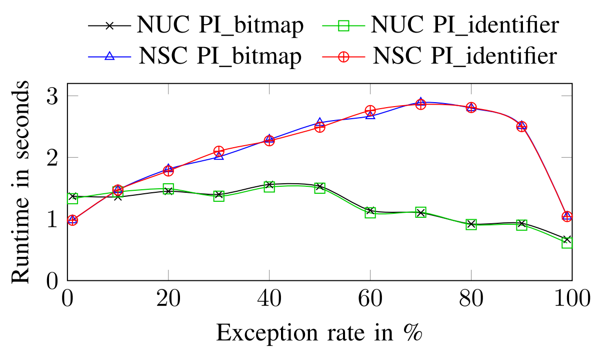

Fig. 6. Patchindex creation time on dataset with varying exception rate

C. Resume

The evaluation showed the existence of approximate constraints in different use cases and proved the expected performance benefit of PatchIndexes for these use cases. The more fine-grained experiments using a custom data generator with varying exception rates show a performance benefit of both NUC and NSC even for very high exception rates. While the bitmap-based approach and the identifier-based approach of realizing the PatchIndex showed nearly similar performance, the bitmap-based approach showed the expected lower memory consumption and is therefore the clearly better choice.

VIII. Conclusion

In this paper, we motivated the need to handle approximate constraints in self-managing databases. We introduced the concept of PatchIndexes for "nearly unique columns" and "nearly sorted columns", handling exceptions to constraints in a common data structure. We provided discovery methods for the discussed constraints that can be easily integrated into existing self-managing databases or auto-administration tools. After discussing different design approaches for the index structure, we presented the index scan mechanism and the integration into sort, join and distinct queries. The evaluation showed use cases for approximate constraints in benchmark representing a retail database and proved the performance benefit when using PatchIndexes.

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. Additionally, the alternatives to the in-memory design described in Section V should be evaluated. Using PatchIndexes comes along with overhead in query execution, mainly caused by additional operators in the query plan and by copying subtrees of query execution graphs. Therefore, we plan to create a cost model covering additional costs of the PatchIndex usage and integrate it into query optimization. Based on this, reasonable values for both *nuc_threshold* and *nsc_threshold* should be defined. Furthermore, we plan to investigate on opportunities the PatchIndex offers for data compression, potentially increasing compression ratios when treating discovered set of patches separately and this way basing compression algorithms on discovered properties of data.

Acknowledgements

This work was funded by the Actian Corp.

References

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: a survey. *The VLDB Journal*, 24(4):557–581, August 2015.

[2] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems*, 28(2):140–174, June 2003.

[3] Arvid Heise, Jorge-Arnulfo Quian-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. Scalable discovery of unique column combinations. *Proceedings of the VLDB Endowment*, 7(4):301–312, December 2013.

[4] Thorsten Papenbrock and Felix Naumann. A Hybrid Approach for Efficient Unique Column Combination Discovery. In Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schning, Melanie Herschel, Jens Teubner, Theo Hrder, Oliver Kopp, and Matthias Wieland, editors, *Datenbanksysteme fr Business, Technologie und Web (BTW 2017)*, pages 195–204. Gesellschaft fr Informatik, Bonn, 2017.

[5] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. Distributed implementations of dependency discovery algorithms. *Proceedings of the VLDB Endowment*, 12(11):1624–1636, July 2019.

[6] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A Machine Learning Approach to Foreign Key Discovery. In *12th International Workshop on the Web and Databases, WebDB 2009, Providence, Rhode Island, USA, June 28, 2009*, 2009.

[7] Erhard Rahm and Hong Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Eng. Bull.*, 23:3–13, January 2000.

[8] Henning Khler, Sebastian Link, and Xiaofang Zhou. Possible and Certain SQL Keys. *Proc. VLDB Endow.*, 8(11):1118–1129, July 2015.

[9] Ziheng Wei, Uwe Leck, and Sebastian Link. Discovery and Ranking of Embedded Uniqueness Constraints. *PVLDB*, 12(13):2339–2352, 2019.

[10] Sebastian Kruse and Felix Naumann. Efficient discovery of approximate dependencies. *Proceedings of the VLDB Endowment*, 11(7):759–772, March 2018.

[11] Y. Huhtala. Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computer Journal*, 42(2):100–111, February 1999.

[12] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter A. Boncz. Super-Scalar RAM-CPU Cache Compression. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 59. IEEE Computer Society, 2006.

[13] Bogdan Ghita, Diego G. Tom, and Peter A. Boncz. White-box Compression: Learning and Exploiting Compact Table Representations. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[14] Michael L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29 – 35, 1975.

[15] Guido Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 476–487, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[16] Transaction Processing Performance Council (TPC). TPC Benchmark DS, Standard Specification, Version 2.8.0. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.8.0.pdf, February 2018.