Detecting Unique Column Combinations on Dynamic Data

Ziawasch Abedjan [#1], Jorge-Arnulfo Quiané-Ruiz [*2], Felix Naumann [#3]

[#] *Hasso Plattner Institute (HPI)*
Potsdam, Germany
[1] ziawasch.abedjan@hpi.uni-potsdam.de
[3] felix.naumann@hpi.uni-potsdam.de

[*] *Qatar Computing Research Institute (QCRI)*
Doha, Qatar
[2] jquianeruiz@qf.org.qa

***Abstract*—The discovery of all *unique* (and *non-unique*) column combinations in an unknown dataset is at the core of any data profiling effort. Unique column combinations resemble candidate keys of a relational dataset. Several research approaches have focused on their efficient discovery in a given, static dataset. However, none of these approaches are suitable for applications on dynamic datasets, such as transactional databases, social networks, and scientific applications. In these cases, data profiling techniques should be able to efficiently discover new uniques and non-uniques (and validate old ones) after tuple inserts or deletes, without re-profiling the entire dataset. We present the first approach to efficiently discover unique and non-unique constraints on dynamic datasets that is independent of the initial dataset size. In particular, SWAN makes use of intelligently chosen indices to minimize access to old data. We perform an exhaustive analysis of SWAN and compare it with two state-of-the-art techniques for unique discovery: Gordian and Ducc. The results show that SWAN significantly outperforms both, as well as their incremental adaptations. For inserts, SWAN is more than 63x faster than GORDIAN and up to 50x faster than DUCC. For deletes, SWAN is more than 15x faster than GORDIAN and up to 1 order of magnitude faster than DUCC. In fact, SWAN even improves on the static case by dividing the dataset into a static part and a set of inserts.**

I. INTRODUCTION

Many emerging applications produce very large datasets at fast rates. Examples of such dynamic data include social networks, scientific measurements, but also traditional transactions. As the amount of data produced by applications continues to grow, the need for a better understanding increases too. Knowing the structure and properties of such datasets is crucial for data integration, data analytics, query optimization, and many further applications. In this context, *data profiling* is emerging as a distinct research area to address the challenges of simple data analytical tasks on large and dynamic datasets [1].

A fundamental task of data profiling is the discovery of unique and non-unique column combinations. Unique column combinations (*uniques*) are column combinations with no duplicate value combination resembling key candidates. In contrast, non-unique column combinations (*non-uniques*) contain at least one duplicate value combination that resemble partial duplicates, i.e., tuples having some column values in common. Overall, uniques and non-uniques are useful for many tasks in the area of data management, such as data modeling, indexing, query optimization, and anomaly detection [2]. Furthermore, uniques and non-uniques represent data-driven rules and constraints of the data. Knowing these constraints supports data analytical tasks. For example, in the life sciences, uniques provide insights about unique protein sequences while knowledge of non-uniques provide insights about re-occurring protein patterns [3]. Furthermore, one can leverage uniques for the discovery of functional and inclusion dependencies [4].

Discovering uniques and non-uniques is a hard problem: it is NP-hard in the number of columns and sub-quadratic in the number of rows. For instance, on a dataset with 100 columns, a brute-force algorithm has to scan the table for all $2^{100}-1$ combinations to discover all uniques and non-uniques. Some existing techniques already tackle the problem of unique discovery for a given dataset in a more efficient manner [2], [5], [6]. All these techniques benefit from the observation that supersets of uniques are also uniques and that subsets of non-uniques are also non-uniques. Thus, these techniques focus on discovering the set of minimal uniques (all of its subsets are non-unique) and the set of maximal non-uniques (all of its supersets are unique) in order to significantly prune the search space. However, all existing unique discovery techniques are not suitable for dynamic data. i.e., scenarios where new data arrives or existing data is removed.

Indeed, discovering uniques and non-uniques over dynamic data is a necessity in several different fields. Query optimisation, data quality monitoring, and reactive duplicate detection are just few examples where *incremental* unique discovery (i.e., over dynamic data) is crucial. For example, many organizations can identify critical datasets that should be of high quality, such as customer-relationship data (master data management) and inventory data. Thus, when monitoring data quality, it is crucial to update meta-data (e.g., uniques and non-uniques) frequently in order to recognize and rectify potential problems as soon as possible. An incremental approach is the

[1]Research performed at QCRI.

TABLE I
EXAMPLE OF PERSONS RELATION INSTANCE

Tuple ID	Name	Phone	Age
$(tuple_1)$	Lee	345	20
$(tuple_2)$	Payne	245	30
$(tuple_3)$	Lee	234	30
$(insert_1)$	Payne	245	31

most efficient way to keep these meta-data up-to-date after the arrival or deletion of data.

Motivating Example. Let us illustrate the problem of current unique discovery techniques via an example. Consider the dataset in Table I. In this example, we have two minimal uniques: {Name, Age} and {Phone}. Accordingly, we have the maximal non-uniques {Name} and {Age}. Now, assume the following two cases:

(1) *Insert* – A new tuple, with the values *(Payne, 245, 31)*, arrives (see last tuple in Table I). Given this inserted tuple, we need to compare whether any minimal unique is violated. Thus, we compare *(Payne, 31)* to all values in {Name, Age} and *245* to all values in {Phone}, which are the minimal uniques. After performing these two uniqueness check, we discover that {Phone} is not unique anymore. As a result, {Age, Phone} is a new minimal unique and {Name, Phone} is a new maximal non-unique, subsuming the previous maximal non-unique {Name}.

(2) *Delete* – The first tuple *(Lee, 234, 30)* is removed. Now we have to check all values in *Name* and in *Age* for duplicate values. This is because we have two maximal non-uniques: *Name* and *Age*. After these checks we discover that *Name* and *Age* are now uniques (excluding *insert*$_1$ of course), resulting into only three minimal uniques for the entire dataset: *Name*, *Phone*, an *Age*.

On the one hand, we observe in this example that new data (i.e., new tuples) can cause new duplicates to appear and hence previously discovered uniques might not be unique anymore. Removing data, on the other hand, can cause existing duplicates to disappear, which might turn some previously discovered non-uniques into new uniques. Thus, it is necessary to detect the new minimal uniques and maximal non-uniques every time a dataset changes. However, current techniques have to profile the changed dataset entirely in order to detect the new minimal uniques and maximal non-uniques. Indeed, profiling the entire dataset hurts performance significantly since an initial dataset is typically several orders of magnitude bigger than the size of a change in the dataset. In these scenarios, data profiling techniques should be able to efficiently discover the new uniques and non-uniques after tuple inserts or deletes, without re-profiling the entire dataset.

Research Challenges. Leveraging previously discovered uniques and non-uniques is crucial to avoid entirely re-profiling large datasets every time a dataset changes. However, leveraging such a knowledge is challenging for several reasons. First, one has to check whether any unique or non-unique constraint has been changed. Doing this consumes a lot of time, because typically one has to validate a quite large number of uniques on the entire dataset. Second, having identified a set of changed unique and non-unique constraints, one has to traverse a huge search space to find the new unique and non-unique constraints. Third, in the quest for new unique and non-unique constraints, one has to validate each of the candidates in the search space. To perform such a validation, one should depend on the size of the changes and not on the size of the complete dataset.

Contributions. In this paper, we propose SWAN, the first approach for unique and non-unique discovery on dynamic data, i.e., on datasets that are continuously changing. SWAN is part of the Metanome data profiling project (www.metanome.de). In summary, we make the following major contributions:

(1.) We model the unique and non-unique discovery on dynamic data by considering two common workloads: inserts and deletes (Section II).

(2.) We present an approach to deal with inserts that depends on the number of inserted tuples. In particular, we propose an algorithm to select a small set of indexes that allows SWAN to efficiently detect changes on existing uniques (Section III).

(3.) We propose an approach to deal with deletes. Our approach can detect changes in maximal non-uniques in a few milliseconds (Section IV).

(4.) We present an exhaustive evaluation of SWAN and compare it with two baseline systems and their respective incremental adaptations, illustrating the high superiority SWAN over the baseline systems in dynamic data scenarios. Furthermore, we show that by transforming a static dataset into a dynamic one, SWAN can process very large datasets that cannot be processed by any state-of-the-art algorithm (Section V).

II. UNIQUES AND NON-UNIQUES ON DYNAMIC DATA

We now introduce the concepts of uniques and non-uniques and define the problem of unique discovery on dynamic data. Then, we briefly discuss the overall architecture of the SWAN system for dealing with inserts and deletes.

A. Problem statement

Given a relation R with a relational instance r, a *unique column combination* (unique) is a set of columns $K \subseteq R$ whose projection on r contains only unique value combinations. Analogously, a set of columns $K \subseteq R$ is a *non-unique column combination* (non-unique), if and only if its projection on r contains at least one duplicate value combination:

Definition 1 (Unique): A column combination $K \subseteq R$ is a *unique* (UC), iff $\forall r_i, r_j \in r, i \neq j : \ r_i[K] \neq r_j[K]$.

Definition 2 (Non-unique): A column combination $K \subseteq R$ is a *non-unique* (NUC), iff $\exists r_i, r_j \in r, i \neq j : \ r_i[K] = r_j[K]$.

Indeed, each superset of a unique is also unique [1] while each subset of a non-unique is a non-unique. Therefore, we can reduce all the effort of discovering all uniques and non-uniques to the discovery of minimal uniques and maximal non-uniques as defined in the following.

Definition 3 (Minimal Unique, MUCS): A column combination $K \subseteq R$ is a MUC, iff $\forall K' \subset K : K'$ is a NUC.

[1]In literature one often refers to the terms key and superkey. A key is a unique that was explicitly chosen while designing a table

Definition 4 (Maximal Non-Unique, MNUCS*):* A column combination $K \subseteq R$ is a MNUC, iff $\forall K' \supset K : K'$ is a UC.

Discovering a minimal unique of size $k \leq n$ has been shown to be NP-complete [7]. To discover all minimal uniques and maximal non-uniques of a relational instance, in the worst case, one has to visit all subsets of the given relation, no matter the strategy (breadth-first or depth-first) or direction (bottom-up or top-down). Thus, we can clearly see that the discovery of all MUCS and MNUCS of a relational instance is an NP-hard problem and that even the solution set can be exponential [8]. Having a relation instance of size n, there can be $\binom{n}{\frac{n}{2}} \geq 2^{\frac{n}{2}}$ MUCS in the worst case, as all combinations of size $\frac{n}{2}$ can be minimal uniques.

When datasets change over time, the set of minimal uniques and maximal non-uniques might change. A new tuple might create new duplicates that change existing uniques into non-uniques. A naive approach to detect the new minimal uniques and maximal non-uniques would compare each new tuple by looking for duplicates on the unique projections. Formally, given a relational instance r, a new tuple t', and the set of minimal uniques MUCS, one has to check $\forall K \subset$ MUCS, if $\exists\, t_i \in r \mid t'[K] = t_i[K]$. For such changed minimal uniques, one has to start comparing t' and t_i with regard to all $K' \supset K$.

Analogously, a removed tuple t_i might change existing non-uniques into uniques. Thus, one has to check whether existing maximal non-uniques $K \in$ MNUCS are affected. Basically, one has to check whether $r[K] \setminus t_i[K]$ still contains duplicate values as defined in Definition 4. If so, K is still a non-unique. Otherwise, one has to check whether subsets of the affected maximal non-uniques are also affected by the removal of t_i.

Therefore, updating the existing unique constraints after a new or removed tuple appeals for processing the complete dataset, i.e., input dataset and the incremental dataset (inserts and deletes) together. So, the challenge in discovering uniques and non-uniques incrementally is: *How to efficiently update the sets* MUCS *and* MNUCS *within a short period of time and without processing the whole input dataset?*

B. *The* SWAN *System: Overview*

We propose SWAN, a system for unique and non-unique discovery on dynamic data. SWAN maintains a set of data structures (indexes) to efficiently find the new sets of minimal uniques and maximal non-uniques after a bunch of inserts or deletes. SWAN is composed of two main components: the *Inserts Handler* and the *Deletes Handler*. The Inserts Handler takes as input a set of inserted tuples, checks all minimal uniques for uniqueness, finds the new sets of minimal uniques and maximal non-uniques, and update the repository of minimal uniques and maximal non-uniques accordingly. Analogously, the Deletes Handler takes as input a set of deleted tuples, searches for duplicates in all maximal non-uniques, finds the new sets of minimal uniques and maximal non-uniques, and updates the repository accordingly. Notice that for each task, we use special data structures that facilitate the detection of new or removed duplicates. In the following two sections, we discuss these two components in more detail.

III. PROCESSING INSERTS

Remember that when new tuples arrive, previously discovered uniques and minimal uniques might not be valid anymore, as the new tuples might create duplicate values for those combinations. Furthermore, any change to the set of minimal uniques also affects the set of maximal non-uniques [6]. In this section, we discuss how SWAN deals with inserted tuples to find all uniques and non-uniques. In particular, we show the challenges to detect changes in the original set of minimal uniques and to discover the new set of minimal uniques and maximal non-uniques. In the following, we first give an overview of the inserts-workflow of SWAN to handle inserts. We then discuss how to efficiently detect duplicates and present our index-based approach to verify a set of minimal uniques. Finally, we show how SWAN efficiently updates the set of minimal uniques and maximal non-uniques based on the detected changes.

A. *Inserts-Workflow Overview*

Algorithm 1 illustrates the overall workflow of SWAN when handling a batch of inserts. The input parameters of the algorithm are the set of minimal uniques MUCS, the set of maximal non-uniques MNUCS, and the set of newly inserted tuples T. The sets MUCS and MNUCS can be obtained by any holistic approach (e.g., GORDIAN [2] or DUCC [6]) for unique discovery when uploading the initial dataset for the first time. Overall, the algorithm has three main phases:

(1.) SWAN compares the inserts to the initial dataset: for each minimal unique SWAN retrieves all tuple IDs that might contain duplicate values (Line 3). If SWAN retrieves no tuple IDs, we can conclude that the new inserts did not create any duplicate for the current minimal unique. This means that the column combination is still a minimal unique. However, if SWAN retrieves some tuple IDs, SWAN then stores them along with the minimal unique in the data structure *relevantLookUps* to handle them later (Line 5). As we already know that all other tuples contain distinct values for the current minimal unique, we also know that the projection of any superset of the minimal unique will be unique on those tuples. Hence, SWAN considers only the tuples that might contain duplicates for the current minimal unique. It is worth noting that depending on the index structure used by *retrieveIDs*, the tuple IDs might also correspond to tuples with partial duplicates with regard to the minimal unique. Those tuples will be discarded later when having the tuple values at hand.

(2.) Once all relevant tuple IDs have been collected by SWAN for all minimal uniques, SWAN computes the union of all IDs and retrieves in one run all relevant tuples by a mix of random accesses and sequential scans of the initial dataset. For this, SWAN uses a sparse index that maps a tuple ID to the byte offset where the tuple resides in the initial dataset (Line 6).

(3.) Once SWAN retrieves all relevant tuples, it uses a duplicate manager *dManager* to group the retrieved tuples and inserted tuples with regard to the corresponding minimal uniques (Line 7). A *duplicate group* is a set of tuples that have the same

Algorithm 1: HandleInserts()

```
Data: Inserted tuples $T$, MNUCS, and MUCS
Result: New MNUCS and MUCS
1 relevantLookUps ← new Map;
2 for $U \in$ MUCS do
3     tupleIds ← retrieveIDs($U$, $T$);
4     if !tupleIds.isEmpty() then
5         relevantLookUps.put($U$, tupleIds);
6 tuples ← sparseIdx.retrieveTuples(relevantLookUps);
7 dManager ← new DuplicateManager(tuples, $T$);
8 return findNewUniques(dManager, MUCS, MNUCS);
```

value combination when projecting the corresponding minimal unique. This partitioning reduces the effort to discover the new minimal uniques in the last step *findNewUniques*.

In the following, we describe how to discover changed uniques by explaining *retrieveIDs()* and the used index structures. Finally we describe *findNewUniques()* in more detail.

B. Checking Uniques

According to Definition 1, all tuples of a dataset have distinct value projections for each minimal unique. The projection of a minimal unique on a newly inserted tuple however might contain a value combination that already exists in the initial dataset. In that case, the uniqueness of the previously identified unique and of its supersets does not hold anymore. If a new minimal unique exists, it must be a superset of the previous one. Thus, when a minimal unique becomes non-unique because of an insert, we have to check all supersets for uniqueness. We can limit the verifications to tuples with duplicates in the invalidated minimal unique. All remaining tuples can be ignored, because they contain unique values for the minimal unique. In order to efficiently identify duplicates among inserted tuples and the initial dataset we use an index-based approach. In the following, we describe the challenge in discovering duplicates and their tuple IDs and present our index-based approach to retrieve the relevant tuple IDs.

Avoiding pairwise tuple comparisons. Given a relation R with an instance r, a new tuple t, and a minimal unique U holding on r, we have to compare $t[U]$ to all $t_i[U]$ with $t_i \in r$. Consider again Table I. In that example, we have two minimal uniques: {Phone} and {Name, Age}. To identify changes in the dataset we have to compare *(Payne, 31)* to all values in {Name, Age} and *245* to all values in {Phone}. For the latter, we discover that the value already exists for the second tuple. So, {Phone} is not a unique anymore. Thus, we have to check whether any superset of {Phone} in the set of attributes qualifies as new minimal uniques. Hereby, we only need to compare the tuple containing 245 with the inserted tuple. As both tuples have the same value for *Name*, {Name, Phone} is also not a unique and has to be extended by the attribute *Age*. However, as both tuples differ for *Age* {Age, Phone} must be a unique and in this case also a minimal unique. We do not need to check all other tuples for {Age, Phone}, because we know that those are already unique with regard to {Phone}.

When having a batch of tuples T and a minimal unique U, we have to compare all tuples $t_j \in T$ to each tuple $t_i \in r$. Additionally, we have to look for duplicates in $T[U]$. In that case, we can group tuples $t_j, t_k \in T$ with $t_j[U] = t_k[U]$. We can skip a group $G \subseteq T$ as soon as we discover a match in $r[U]$, because $r[U]$ contains only unique value combinations. Nevertheless, in the worst case, we have to do a fullscan on the initial dataset and $|r|$ times a fullscan on the grouped inserted tuples. This is why SWAN uses indexes that map values in $K \subseteq U$ to the corresponding tuple ids. This way SWAN can discover all duplicates with only one fullscan of T per index.

Index-based retrieval of tuple IDs. Algorithm 2 shows how SWAN identifies and retrieves relevant tuple IDs for a minimal unique. SWAN receives a minimal unique U and the set of inserted tuples T as input parameters. Our indexes may cover a minimal unique completely or only partially. Furthermore, a minimal unique may be covered by multiple disjoint indexes. In that case, SWAN performs a look-up on each index and intersect the results. Thus, SWAN might use an index for multiple minimal uniques. To avoid repeating the grouping of inserted tuples and index look-ups, SWAN caches look-up results as well as intersection results. In line 1 we check whether the look-up of any subset *CC* of the current minimal unique has been performed before. In that case in Line 4 we directly retrieve the cached results. If the cached look-up result is empty, SWAN can already stop the uniqueness check for U, because U will stay unique. Otherwise, SWAN retrieves all existing indexes that cover U (line 7) except the cached indexes of *CC*. For each index *idx*, we add the associated column combination to *CC*. If no cached results were retrieved and *lookUpResults* is empty, we then perform the first index look-up for U (Line 11).

Notice that SWAN groups the inserts by the distinct value projections of *idx.getCC()* in order to avoid multiple scans of the inserts and unnecessary look-ups. If *lookUpResults* is not empty SWAN performs a modified look-up on the index. This way, SWAN simulates an intersection by considering only those projections of *idx.getCC()* on the inserted tuples T that were included in the previous look-up results. Afterwards, SWAN caches the accumulated index columns *CC* and the corresponding results. If the look-up results are empty, SWAN can return the empty set *lookUpResults*. Otherwise, SWAN moves to the next index *idx*. SWAN finishes at latest when all relevant indexes for U have been used and returns the final non-empty set of *lookUpResults*.

C. Minimal set of Indexes: Avoiding Full Scans

The reader might think that the best way to index a minimal unique would be to create a multicolumn index that cover the entire minimal unique. This way one would perform a single look-up per distinct value projection and minimal unique. However, datasets usually contain hundreds of minimal uniques in practice and creating such a large number of indexes is expensive (computational- and storage-wise). Furthermore, one cannot use multicolumn indexes anymore

Algorithm 2: RetrieveIDs()

```
Data: Inserted tuples $T$, minimal unique U
Result: tupleIds
CC ← getLargestCachedSubset(U));
lookUpResults ← new List;
if !CC.isEmpty() then
    lookUpResults = tupleIdCache.get(CC);
    if lookUpResults.isEmpty() then
        return lookUpResults;
indexes ← getIndexes(U \ CC);
for idx ∈ indexes do
    CC.add(idx.getCC());
    if lookUpResults.isEmpty() then
        lookUpResults ← idx.lookUpIds([T]);
    else
        lookUpResults ← idx.lookUpAndIntersectIds(lookUpResults);
    cache(CC, lookUpResults);
    if lookUpResults.isEmpty() then
        return lookUpResults;
return lookUpResults;
```

as soon as a minimal unique loses its uniqueness after an increment of new tuples. Also, indexing all single columns is still very expensive (storage-wise) as a relation might consist of hundreds of columns. Updating and maintaining all the indexes is still too costly.

Therefore, SWAN takes a different approach: SWAN indexes a small subset of columns so that all minimal uniques are covered by at least one index. Thereby, SWAN follows a greedy approach based on the frequency of columns among minimal uniques in order to choose the right columns to index. Notice that the frequency of a column among minimal uniques is correlated to its selectivity as columns with many distinct values occur in many minimal uniques. Of course, as indexes might only cover subsets of a minimal unique, the IDs retrieved by those indexes might be only partial duplicates with regard to the projections on the newly arrived tuples. Thus, after the retrieval of actual tuples, SWAN checks the few remaining duplicate groups by verifying the values of the columns without an index.

Algorithm 3 shows how SWAN chooses indexes based on a given set of minimal uniques and the corresponding attributes in the relation R. First, the frequency of each column with regard to its participation among the minimal uniques is retrieved. The most frequent column C_f is added to the set of to be indexed columns K. To choose the next column, system excludes all minimal uniques that contain C_f and retrieves the column frequencies on the remaining minimal uniques. Again, the most frequent column is added to K and SWAN excludes all covered minimal uniques for choosing the next column. This process is repeated until all minimal uniques have been covered by at least one column.

Algorithm 3: SelectIndexAttributes()

```
Data: MUCS, relation R
Result: Columns to be indexed K
frequencies ← new Map;
for $C \in R$ do
    frequencies.put($C$, getFrequency($C$, MUCS));
$C_f$ ←frequencies.getMostFrequentColumn());
$K$.add($C_f$);
remainingMUCS ← MUCS $\setminus\{U|U \in$ MUCS $\wedge C_f \in U\}$;
while !remainingMUCS.isEmpty do
    frequencies.clear();
    for $C \in R$ do
        frequencies.put($C$, getFrequency($C$, remainingMUCS));
    $C_f$ ←frequencies.getMostFrequentColumn());
    $K$.add($C_f$);
```

D. Additional Indexes: Speeding-Up SWAN

Although the minimal set of indexes helps SWAN to significantly reduce the number of tuples, we still can speed up the look-up and verification process if we reduce the number of false positives. Hence, it is also desirable to create extra indexes that allow SWAN to reduce the number of false positives. However, choosing more columns to index is not always beneficial. Imagine a scenario with four minimal uniques $\{A, B\}$,$\{A, C\}$, $\{A, D\}$, and $\{C, D\}$. Our index selection approach creates the indexes I_A and I_C on the columns A and C. While for an inserted tuple t' and the minimal uniques $\{A, B\}$ and $\{A, D\}$ the index I_A retrieves the same set of tuple IDs $T(I_A)$, for the minimal unique $\{A, C\}$ the set of tuples may be smaller as we can calculate the intersection $T(I_A) \cap T(I_C)$. However, globally this intersection does not save us any reduction with regard to the number of tuples to be retrieved as we need the larger set $T(I_A)$ for the uniques $\{A, B\}$ and $\{A, D\}$ and even $T(I_C)$ for the unique $\{C, D\}$. In total, our example leads to the retrieval of $T(I_A) \cup T(I_C)$. So, the motivation is to reduce the number of tuples to be retrieved. In case we were allowed to create another index, we could choose between B, and D. By choosing B, we do not reduce the amount of retrieved tuples in total. As $\{A, D\}$ is still only covered by I_A, $t(I_A)$ will be retrieved anyway and we still have to retrieve $T(I_A) \cup T(I_C)$. But if we choose to index D, there is a chance that we reduce the tuples that would have been retrieved by C, because we can effectively reduce $T(I_C)$ to $T(I_C \cap I_A) \cup T(I_C \cap I_D)$ knowing that $|T(I_C \cap I_A) \cup T(I_C \cap I_D)| \leq |T(I_C)|$.

Algorithm 4 illustrates how SWAN chooses more indexes based on the initial set of index columns K and a user-defined quota δ that limits the total number of columns to be indexed.

First, for each column $C \in K$, SWAN strives to find the best possibility to cover that column among all minimal uniques without exceeding the given quota $\delta > |K|$. SWAN applies Algorithm 3 to the modified minimal unique set as presented in Line 4. The modified set consists of all minimal uniques

Algorithm 4: addAdditionalIndexAttributes()

```
Data: MUCS, relation R, initial indexes $K$, quota $\delta$
Result: Columns to be indexed K
1  $I_K$ ←createIndexes($K$);
2  coveringIndexes ← new Map; solutions ← new Map;
3  for $C \in K$ do
4      containingMUCS
       $\leftarrow \{U \setminus \{C\} : U \in$ MUCS $\wedge (U \cap K) == \{C\}\}$;
5      $K_C \leftarrow$ selectIndexAttributes(containingMUCS,R);
6      if $|K_C| \leq \delta$ then
7          coveringIndexes.put($C$, $K_C$);
8  for combination $C_1, C_2, ..C_k \in$ coveringIndexes.keySet() do
9      if $|K_{C_1} \cup K_{C_2} .. \cup K_{C_k}| \leq \delta$ then
10         solutions.put($C_1 \cup C_2, .. \cup C_k$,
           $K_{C_1} \cup K_{C_2} .. \cup K_{C_k}$);
11 solutions.removeRedundantCombinations();
12 $K_0 \leftarrow$ combLowestSelectivity(solutions.keySet(), $I_K$);
13 $K$.add(solutions.get($K_0$));
14 return $K$;
```

that contain the column C, but no other indexed column. As the function *selectIndexAttributes* generates the smallest set of columns that covers all of these minimal uniques in a greedy way we have to make sure that the column C, which covers all of them, is removed beforehand. Second, SWAN generates all possible combinations of columns $C_1, C_2, ..C_k \in K$ that can be covered by a set of covering attributes and create the union of the corresponding covering attributes $K_{C_1}, K_{C_2} .., K_{C_k}$. If the size of the union is below δ we store the solution (Line 10). In a last step, we choose the solution that covers least selective combination $C_1, C_2, ..C_k$. We define the selectivity $s(C_i)$ of an index I_{C_i} on a relational instance r as follows:

$$s(C_i) = \frac{cardinality(C_i)}{|r|}$$

The cardinality of a column denotes the number of distinct values of this column. Accordingly a primary key column has the cardinality of $|r|$ and a selectivity of 1. To identify the selectivity of the set of columns $C_1, C_2, ..C_k \in K$ we apply the following formula that corresponds to union probability:

$$s(C_1, C_2, ..C_k) = 1 - ((1 - s(C_1) \cdot (1 - s(C_2) \cdot \ldots \cdot (1 - s(C_k))$$

After several inserts the indices just need to be updated by adding the new values and row ids. If an index contains the new value, we append the tuple ID to the value's ID list; otherwise, we create a new key-value-pair. As inserts may change the set of minimal uniques only in a way that the new minimal uniques are supersets of previous minimal uniques, we do not need to create new indexes. An index that covers a minimal unique $U \subseteq R$ also covers any superset $U' \supset U$. This property does not hold after deletes, because subsets of previous minimal uniques may become new minimal uniques. In that case, our index selection approach should be applied again to check whether new indexes should be created.

E. Finding new Minimal Uniques and Maximal Non-Uniques

Once SWAN identifies that a minimal unique changed after a set of inserts, SWAN has to traverse all supersets of the changed minimal unique to discover all new minimal uniques. Here, SWAN reduces the effort for varifying a combination on the complete dataset to the duplicate groups that have been identified via our indexes.

Algorithm 5 illustrates how SWAN discovers new possible uniques and non-uniques based on the tuples stored in the duplicate manager along with the previous set of MUCS and MNUCS. For each unique U from our previous set, SWAN retrieves the corresponding duplicate groups (Line 2). Notice that SWAN indexes might not completely cover all columns of a minimal unique. So, SWAN has to remove all those partial duplicates from the duplicate groups (Line 3). For uniques that did not change, the groups are empty. Thus, SWAN simply adds them to the set *newUniques*. Otherwise, SWAN knows that U is non-unique and generates all relevant supersets by adding single columns to the combination U (lines 8). For each duplicate group, SWAN checks whether the candidates are unique. If not, SWAN expands each candidate with new columns and changes the set of candidates for the next duplicate group (Line 11). Having processed all previous minimal uniques, SWAN simplifies the sets of newly discovered minimal uniques and maximal non-uniques, MUCS and MNUCS, to remove redundant supersets and subsets respectively (Line 23). Notice that as an insert cannot turn a non-unique to a unique, the set of maximal non-uniques can only change if a unique superset of a maximal non-unique turns non-unique. In that case we can remove all its subset combinations from MNUCS.

IV. PROCESSING DELETES

In contrast to the effect of new tuples, when removing tuples d from a relational instance r, non-unique combinations can turn into unique. This is because deleting tuples may lead to the removal of duplicate values among some columns. Therefore, deleting tuples may affect both the set of minimal uniques as well as the set of maximal non-uniques. A maximal non-unique may turn into a unique. And if a non-unique turns into a unique combination any minimal unique that is a superset of that combination is not minimal anymore. In the following we present SWAN's workflow to manage deletes, and our strategies to check non-uniques efficiently.

A. Deletes-Workflow Overview

Algorithm 6 illustrates the overall workflow of SWAN after deletion of tuples. In contrast to the inserts-workflow, after a batch of deletes SWAN has to look for changes in the set of non-uniques. So, SWAN first stores all minimal uniques MUCS into the data structure *UGraph* (Line 3). The data structures *UGraph* (unique graph) and *NUGraph* (non-unique graph) assure that SWAN can omit redundant combinations immediately as soon a new minimal unique or maximal

Algorithm 5: findNewUniques()

```
Data: duplicate manager dManager, mUcs, mnUcs
Result: New mUcs and mnUcs
1  for $U$ : mUcs do
2      groups ← dManager.getGroups($U$);
3      removePartialDuplicates(groups);
4      if groups.isEmpty then
5          newUniques.add($U$) ;
6          continue ;
7      candidates ← new List ;
8      for $C \in R \setminus U$ do
9          candidates.add($U \cup \{C\}$);
10     for group : groups do
11         for $K \in$ candidates do
12             candidates.remove($K$);
13             while !isUnique($K$) $\wedge K \neq R$ do
14                 mnUcs.add($K$);
15                 $C$ ← takeCol($R \setminus K$);
16                 $K$ ← check($K \cup C$, group);
17             if $K = R$ then
18                 go to line 23
19             candidates.add($K$);
20         removeRedundant(candidates);
21     newUniques.addAll(candidates) ;
22 mUcs ← newUniques;;
23 removeRedundant(mnUcs,mUcs);
24 return (mUcs,mnUcs) ;
```

non-unique is discovered. A mapping of columns to column combinations enables the fast discovery of previously discovered redundant combinations. Because all non-uniques of a relation are subsumed by the set of maximal non-uniques, SWAN can start the analysis on that set. If a maximal non-unique stays non-unique all of its subsets will also stay non-unique. In that case, SWAN just adds the combination NU to *NUGraph* (Line 6). If the combination turns out to be a new unique, SWAN adds the combination to *UGraph*. Then, SWAN starts to check whether any other subset of NU also turned into a unique combination. Here, SWAN checks recursively in a depth-first manner all subsets of NU and stores the intermediate unique and non-unique discoveries into *UGraph* and *NUGraph*. As maximal non-uniques may have overlaps in their subsets, SWAN also uses the *UGraph* and *NUGraph* structures to avoid unnecessary non-uniqueness checks. If *NUGraph* contains a superset of a combination to be checked, SWAN can prune the combination and all of its subsets. If *UGraph* contains a subset $K \subset NU$, SWAN can reduce the search space to all subsets $K' \subseteq NU$ where K' is not a superset of K.

B. Checking Non-uniques

To identify whether a previous non-unique N on a relational instance r is still non-unique we need to know whether $r'[N]$

Algorithm 6: HandleDeletes()

```
Data: Relation $R$, Ids of Deleted Tuples $D$, mnUcs, and mUcs
Result: New mnUcs and mUcs
1  UGraph ← empty graph; NUGraph ← empty graph;
2  for $U \in$ mUcs do
3      UGraph.add($U$);
4  for $NU \in$ mnUcs do
5      if isStillNonUnique($NU$, $D$) then
6          NUGraph.add($NU$);
7      else
8          UGraph.add($U$);
9          checkRecursively($NU$, $D$, UGraph, NUGraph);
10 mUcs ← UGraph.getminimalUniques();
11 mnUcs ← UGraph.getmaximalUniques();
12 return (mUcs,mnUcs);
```

with $r' = r \setminus d$ still contains a duplicate value or not. In other words, we need to know whether the current tuple deletion has removed all duplicates from $r'[N]$ or not. A straightforward approach would follow a sort-based or hash-based approach to discover duplicates in $r'[N]$. This approach would in the worst case lead to a fullscan of r'. Furthermore, if N turns into a unique we have to do the complete duplicate detection approach again for all subsets $N' \subset N$ with $|N'| = |N| - 1$. We would unnecessarily scan unique values among N multiple times. In contrast, SWAN limits the search space to duplicate groups in N, only. Here, SWAN follows the spirit of approaches such as [9], [10], which use inverted indexes called position list indexes (PLIs) per column.

Position list indexes. A PLI for a column K is a list of position lists, where each position list contains tuple-IDs that correspond to tuples with the same value combination in K. Our approach uses PLIs to efficiently discover whether a combination turned to a unique or not. The PLIs per column can be obtained during the initial run of unique discovery when the data is scanned. The indexes are much smaller than the actual columns, because they store only IDs for values that occur more than once. To obtain the PLI for a column combination K we only have to cross-intersect the PLIs of all columns $C \in K$. The PLI of a non-unique K obviously contains at least one PL with a duplicate pair. To update a PLI after a delete we have to remove the existing IDs of all removed tuples from the PLI. Remember, only IDs of duplicate values have been stored in the first place, and if the removal of an ID from a PL changes its cardinality to 1, the PL can be omitted. So, we simply have to check whether there are still PLs available for K in order to know if K remains non-unique.

Avoiding complete intersections. In many cases a complete PLI intersection is not necessary. For example, if a deleted tuple contains a unique value for some column $C \in K$, we can already conclude that the deletion of the tuple cannot affect the PLI of K. Furthermore, if we consider only PLIs that contained the deleted tuples and reduce the intersection to those *relevant* PLIs before removing the IDs of deleted tuples, we could already conclude non-uniqueness if the intersection of all relevant PLIs per column result in an empty set of PLIs. In that case the removed tuples did not affect the duplicates in K. If the result contains some PLIs we check whether they contain IDs of removed tuples. If any of the PLIs contains at least two IDs that do not correspond to the set of deleted tuples we can conclude non-uniqueness. Otherwise, the removal of the tuples has affected a set of duplicates in K and we need to check the complete PLI of K.

V. EXPERIMENTS

We now evaluate the performance of SWAN on different datasets to answer the following questions: How well does SWAN deal with different sizes of inserts and deletes? How good is SWAN on large initial datasets? How well does SWAN scale in the number of columns? How efficient are the indexes created by SWAN? Can SWAN behave as a holistic approach on static datasets? To this end, we first describe our experimental setup, datasets and the most relevant baselines. We then present a series of experiments that compares SWAN to baselines in dealing with incoming new data. Next, we conduct experiments with very high increment sizes and analyze the ability of SWAN to deal with static data. Then, we show how SWAN performs in scenarios where data is deleted.

A. Setup

Server & Datasets. We use the following server for all our experiments: two 2.67GHz Quad Core Xeon processors; 16GB of main memory; 320GB SATA hard disk; Linux CentOS 5.8 64-bit; 64-bit Java 7.0. We use two real-world datasets and one synthetic dataset in our experiments: the North Carolina Voter Registration Statistics (NCVoter) dataset, the Universal Protein Resource (Uniprot, www.uniprot.org) dataset, and the TPC-H lineitem relation (TPC-H). The NCVoter dataset contains non-confidential data about 7,503,575 voters from the state of North Carolina. This dataset is composed of 94 columns and has a total size of 4.1GB. The Uniprot dataset is a public database of protein sequences and functions. Uniprot contains 539,165 fully manually annotated and curated records and 223 columns, and has a total size of 1GB. The synthetic *lineitem* table (with scale-factor 3) has 16 columns. For all datasets the number of unique values per column approximately follows a Zipfian distribution.

Systems. We use GORDIAN [2] and DUCC [6] as baselines. GORDIAN is a row-based unique discovery technique based on non-unique discovery on a prefix tree. Among column-based approaches we compare to DUCC, which combines aggressive pruning through random walk and PLI representation of columns. We further conducted an experiment that includes the runtime of a well-known commercial DBMS for adding new tuples to a table with unique constraints.

We made a best-effort java implementation of GORDIAN according to the description given in [2]. For fairness reasons we adapted to deal with inserts (GORDIAN-INC): We provided GORDIAN with the information about previously discovered maximal non-uniques, because GORDIAN is based on non-unique discovery. We only considered the time frame for adding the inserted tuples into the prefix tree, assuming that the initial dataset is already in the prefix tree. For deletes, we only consider the time for removing tupels from the prefix tree instead of creating a complete new tree. Here, GORDIAN-INC cannot use the previously discovered maximal non-uniques, as they may not be correct after the delete.

We also adapted the original DUCC to deal with deletes (DUCC-Inc) by providing it with previously discovered minimal uniques, removing the subset graph above those uniques from the search space of DUCC. Unfortunately, DUCC could not be adapted the same way for handling inserts. Apriori knowledge of non-uniques that have not been discovered during the original run of DUCC lead to infinite loops of the random walk because of the bottom-up design of DUCC.

B. Dealing with Inserts

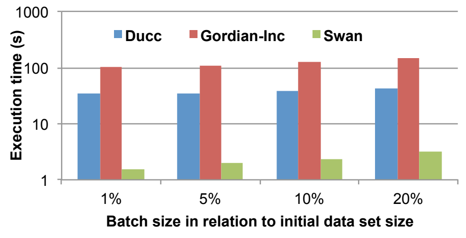
To better analyze the performance of SWAN over incoming data, we perform four different experiments. First, we show how SWAN compares to baselines for different amounts of incoming tuples. Second, we show the runtime behavior of SWAN with regard to the initial dataset size. Third, we evaluate SWAN with different number of columns. Fourth, we evaluate the efficiency of the selected indexes by SWAN. Note, for all these experiments, we present results for DUCC, because one cannot adapt DUCC to deal with new incoming data.

Scaling the batch-size on small datasets. We chose a small sample of 100k tuples per dataset since GORDIAN-INC, the incremental adaptation of GORDIAN, does not finish for larger datasets. We also restrict the number of columns for NCVoter and Uniprot to 40 attributes in order to have a fair comparison with GORDIAN, which does not scale to a larger number of columns on larger datasets.

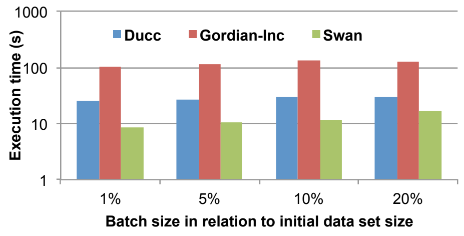
Figure 1a shows the results on the NCVoter dataset. SWAN outperforms both DUCC as well as GORDIAN-INC for all batch sizes. On the smallest batch size of 1k new tuples, SWAN is more than 20x faster than DUCC and more than 63x faster than GORDIAN-INC. We also observe that the runtime of all three systems increases sublinearly with the increment size. Thus, even for a large increment size of 20k tuples, SWAN is still 12x faster than DUCC and 40x faster than GORDIAN-INC.

Figure 1b illustrates the results on the Uniprot dataset. The results are similar to the NCvoter results. SWAN again outperforms both approaches. SWAN is up to 3 times faster than DUCC and up to more than one order of magnitude faster than GORDIAN-INC. Again, the runtime of all three systems increases sublinearly to the the increment size. But, this time the ratio is slightly smaller, because the Uniprot dataset has more duplicates resulting into much more index look-ups for SWAN. For example, having 1k increment SWAN retrieves 97801 tuples (which is nearly the complete dataset), while on the NCVoter dataset SWAN touches 5507 tuples out of 100k.

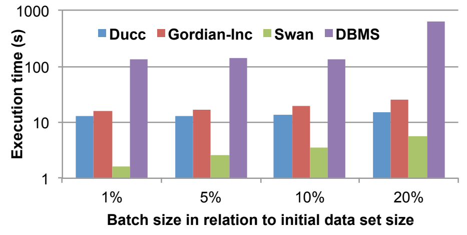
Figure 1c shows the results for the TPC-H dataset. This time we also include the runtime of a commercial database (DBMS-



(a) NCVoter with 100k rows and 40 columns

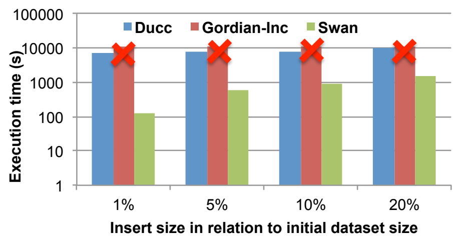


(b) Uniprot with 100k rows and 40 columns

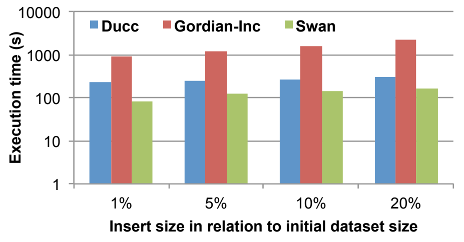


(c) TPC-H with 100k rows and 16 columns

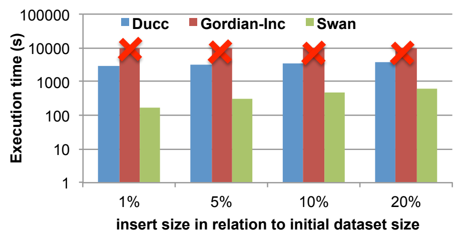
Fig. 1. Scaling the Number tuples per insert batch



(a) NCVoter with 5 millions rows and 40 columns



(b) Uniprot with 400k rows and 40 columns



(c) TPC-H with 5 millions rows and 16 columns

Fig. 2. Scaling the Number tuples per insert batch and larger initial dataset

X)[2]. We see that DBMS-X is by several orders of magnitude slower than SWAN. It is worth noting that DBMS-X needed only 120 ms to add the 20k batch when no constraints were defined. Indeed, this performance gap might also be caused by some DBMS-X related overhead. Regarding the other two baseline systems, the results follow the same pattern as in the previous two datasets results. SWAN is up to one order of magnitude faster.

Scaling the batch-size on large datasets. We now evaluate SWAN with a larger initial dataset. We increased the initial dataset size to 5 million tuples for NCVoter and TPC-H and to 400k tuples for Uniprot. Like previous experiments, we consider 40 attributes for NCVoter and Uniprot and 16 columns for TPC-H.

Figure 2 illustrates the results of these experiments. Overall, we observe that SWAN follows the same behavior as in Figure 1. We observe in Figure 2a that SWAN outperforms DUCC by almost 2 orders of magnitude and GORDIAN-INC by more than 2 orders of magnitude. In fact, we had to abort GORDIAN-INC after 10 hours as it was not even able to update the prefix tree within that time frame. In Figure 2b, we see that the runtime behavior of all systems is quite similar to their runtime on the smaller Uniprot sample (Figure 1b). SWAN outperforms all two baselines significantly. In Figure 2c, we see the high superiority of SWAN: it is up to 15x faster than DUCC for 50k inserts and more than 5x faster for 1 million tuples. Again, we had to abort GORDIAN-INC after 10 hours not being able to update the prefix tree.

Scaling the number of columns. In the previous experiments, we could observe that speed-up ratio of SWAN was comparable for NCVoter and TPC-H although the datasets comprised different number of columns: 40 for NCVoter and 16 for TPC-H. Therefore, we run a another series of experiments

[2]DBMS-X only checks whether new tuples violate the predefined set of 268 minimal uniques, i.e., DBMS-X does not discover new constraints

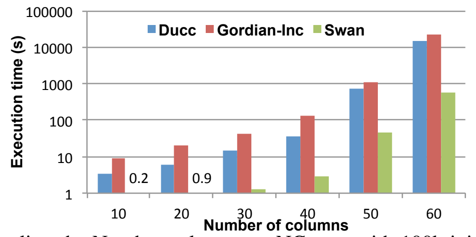
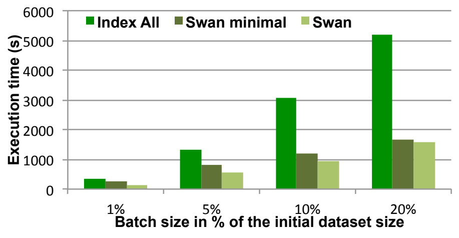


Fig. 3. Scaling the Number columns on NCvoter with 100k initial data size and 10k inserts

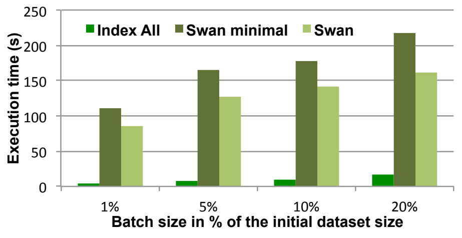
for NCVoter where we vary the number of columns. In these experiments, the amount of inserted tuples corresponds to 10% of the initial dataset, which contains 100k tuples.

Figure 3 illustrates the results. On the projection with 10 columns all systems are quite fast and below 10 seconds. Still, SWAN outperforms both systems by more than one order of magnitude. In fact, this trend stays constant for all projections up to 60 columns, where SWAN outperforms DUCC by more than 20x and GORDIAN-INC by more than 31x. On 70 columns GORDIAN-INC and DUCC could not finish within a time frame of 10 hours even for the initial dataset.

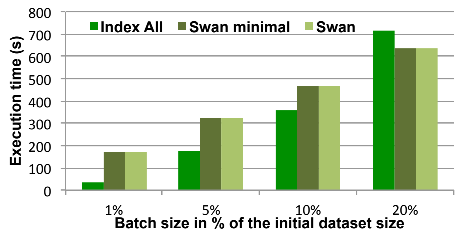
Index Analysis. We now evaluate the efficiency of the indexes created by SWAN. Therefore, we run a series of experiments over all three datasets and consider three variants of SWAN: SWAN with the set of minimal indexes (SWAN minimal), with the complete set of indexes (SWAN), and with an index on each attribute (Index All). We limited the quota for our index selection approach to 20 attributes for NCVoter and Uniprot and to 8 columns for TPC-H. Figure 4a illustrates the results for the NCVoter dataset. Here, SWAN minimal uses 11 indexes, SWAN uses 18 indexes, Index All uses 40 indexes. We observe that SWAN is always faster than SWAN minimal. For realistic increment sizes such as 1%, it is even two times faster with only 7 more indexes. The ratio decreases with the insert size because more indexes always means that more look-ups have



(a) NCVoter with 5 million rows and 40 columns



(b) Uniprot with 400k rows and 40 columns



(c) TPC-H with 5 million rows and 16 columns

Fig. 4. Analysing SWAN with different sets of indexes

to be performed and the batch size defines the runtime of an index look-up. In particular, we observe that Index All is much slower than both SWAN versions, although the index look-ups ensure that only real duplicates have been retrieved.

Figure 4b shows the results for the Uniprot dataset. Overall, we again observe this time the difference between SWAN minimal and SWAN is much smaller. This is because SWAN minimal already uses 17 indexes while SWAN uses one more index (note, quota was set to 20). Still, we see that adding one more index leads to a 1.3x speed-up of SWAN. This time indexing all columns leads to drastic performance boosts. Note on the Uniprot dataset the batch sizes are much smaller than for the NCVoter dataset. So in this specific case the small amount of inserts did not lead to runtime explosion of index look-ups and intersections. Figure 4c illustrates the results for SWAN for the TPC-H dataset. As Algorithm 4 did not propose any additional indices for that data set, SWAN and SWAN minimal use the same set of 6 indices resulting in the same execution time. Moreover, we see in this experiment that indexing all 16 columns only slightly improves the execution time of SWAN for small increment sizes. For the largest batch size indexing all approaches results in more execution time, because of the same reason we gave for the NCVoter dataset.

We also used the index advisor of a commercial database (DBMS-X) to advise indexes according to two different workloads. The first workload contains statements to count the number of distinct values in each minimal unique on the dataset. This workload then resembles the verification of the current minimal uniques. However, the indexes advised by index advisor of DBMS-X were not useful as they did not cover any minimal unique. For example, the index advisor proposed 63 multidimensional indexes for TPC-H, but none of them corresponded to any minimal unique forcing to perform a fullscan. The second workload also included the set of 1k inserts, but in this case the index advisor did not suggest any index. Our approach that is based on single column indexes consumes only milliseconds on the 100k datasets and up to 10 seconds on the large datasets to update all indexes. On larger datasets the update could easily be taken offline though.

C. SWAN *as a Holistic Approach*

Indeed, one can easily model any static scenario as an dynamic one by dividing the dataset into a static initial dataset and one (or more) incremental chunks. For most kinds of

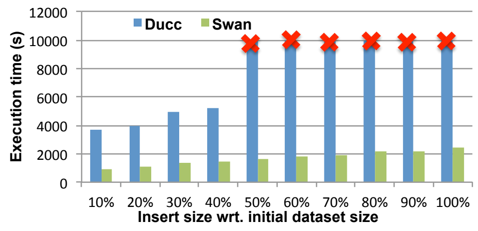


Fig. 5. Scaling the Number tuples per insert batch on TPCH with 16 columns and 5 mill rows.

problems such a design would lose against a holistic approach that is applied on the complete dataset.

As a first series of experiments, we run an experiment where we increase the size of the incremental chunk. The results are depicted in Figure 5. We observe that SWAN significantly outperforms DUCC for any size of the incremental chunks. Up to an increment size of 40%, SWAN is 4x faster than DUCC on average. From an increment of 50%, i.e., from 7,500,000 inserted tuples, we aborted DUCC after 10 hours as it hit the main memory capacity. In contrast, SWAN could finish to process the dataset even for an increment of 100% in almost 30 minutes. In other words, we were able to discover the minimal uniques of a dataset with 10 million tuples much faster than any holistic approach by combining DUCC and SWAN.

We further compare SWAN with DUCC for different numbers of columns (DUCC. Figure 6 shows the results of this series of experiments. For this series of experiments, we take the same data chunks as in Figure 3. We evaluate two different versions of SWAN: one having 100k tuples as initial dataset and 10% incremental data and another having 10k tuples as initial dataset and 100k tuples as incremental data. Note that in contrast to all experiments before, the results we report here for SWAN comprise the runtime of DUCC on the initial sample plus the runtime of the incremental approach to deal with the incremental chunks and the index creation for SWAN.

For up to 30 columns, we observe that the runtime of SWAN with the 10k sample is much faster than SWAN with the 100k sample and DUCC. As the sample data is smaller its static part and the index creation are much faster on the smaller 10k sample. However from 40 columns on SWAN with the 10k sample gets by orders of magnitude slower than DUCC. For these data sets we could observe that the shape and number of the minimal uniques changes drastically from 10k to 100k tuples. On the other hand SWAN with the 100k sample was slightly slower than DUCC for the data sets with 10, 20 and

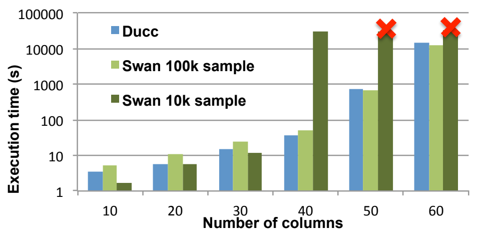


Fig. 6. Holistic SWAN on 110k tuples from NCVoter

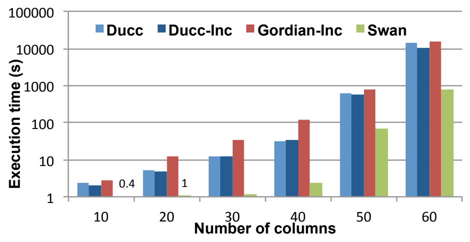


Fig. 8. Scaling the Number columns on NCvoter with 100k initial data size and 10k deleted tuples

30 columns, but starts to slightly outperform DUCC on the datasets with more columns.

D. Deletes

We finally evaluate SWAN in a scenario where tuples are deleted. In this experiments, we additionally compare SWAN with DUCC-INC, the adaptation of DUCC, to deal with dynamic data (deletes only). First analyze the runtime of SWAN dealing with different amounts of deletes. Then we then analyse SWAN with different number of column.

Scaling the number of deleted tuples. Figure 7 shows the results for all three datasets and for different numbers of deleted tuples. We observe in Figure 7a that SWAN outperforms all baseline systems, except for 20% deletes where SWAN is slightly slower than DUCC-INC. This is because as more deletes occur the smaller the dataset is for DUCC and DUCC-INC to analyse. However, having 20% of deletes is an unusual case. In practice, one typically finds less than 1% of deletes. In this case, SWAN is 50x faster than DUCC and more than 8x faster than DUCC-INC. Regarding GORDIAN-INC, we had to abort it, because it again did not finish before 10 hours.

In Figure 7b, we see a similar behaviour as for NCVoter. The results show again the high superiority of SWAN for realistic scenarios, i.e., for a small amount of deletes. SWAN outperforms DUCC and GORDIAN-INC by more than one order of magnitude and DUCC-INC by more than 5x. For 5% deletes, SWAN is still the fastest system: it is 1.3x faster than DUCC-INC. However, SWAN is slightly outperformed by DUCC-INC from 10% deletes.

Figure 7c shows the results for the TPC-H dataset. In these results, we observe a similar pattern as before. SWAN clearly outperforms the baselines systems for 1% deletes and continuously loses its superiority to DUCC-INC when more tuples are deleted. Again, we had to abort GORDIAN-INC, because it did not manage to finish before 10 hours.

Scaling the number of columns. Our last experiment illustrates how the number of columns affects the runtime of SWAN in comparison to baseline systems when removing tuples. We fix the number of deletes to 1% of the initial dataset size, which is realistic in practice. Figure 8 illustrates the results of these experiments. We observe that SWAN significantly outperforms all baselines systems: SWAN is up to more than one order of magnitude faster. In particular, we observe that, until 40 columns, SWAN is able to finish before 5 seconds.

E. Summary

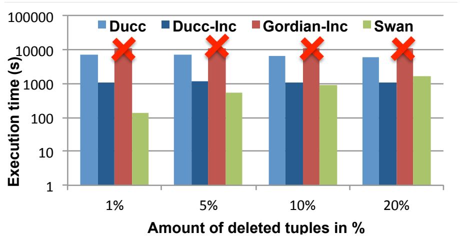
Overall, we observed that SWAN always outperforms state-of-the-art systems significantly over dynamic datasets, e.g., SWAN is one order of magnitude faster than DUCC for 10% inserts. In general, the runtime of SWAN is linear to the increment size. In particular, we observed that the indexes created by SWAN significantly improve the performance towards the minimal set of indexes and that adding more indexes even reduces the runtime of the algorithm. Furthermore, SWAN is able to process very large increments for uniform data and can substitute holistic approaches that are not able to process the whole dataset. Specifically, SWAN enables holistic approaches (in this case DUCC) to achieve what was not possible before, i.e., to find all uniques and non-uniques in datasets with more than 7,500,000 tuples. Finally, the results showed that SWAN is superior to previous baseline systems in realistic scenarios with up to 5% deleted rows. All these results clearly show the high efficiency of SWAN to deal with both inserts and deletes.

VI. RELATED WORK

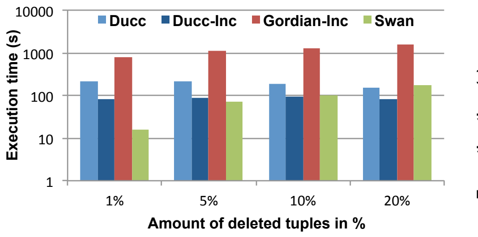
Although knowledge about uniques is fundamental in database management and many other fields (such as bioinformatics and data mining), their automatic discovery has been the focus of surprisingly few research works [2], [5], [6], [11]. There are basically two different classes of techniques in the literature: row-based and column-based techniques. While row-based techniques benefit from the intuition that non-uniques can be detected without considering all rows in a relation, column-based techniques benefit from previously discovered uniques to prune the search space.

A prominent approach unique discovery is GORDIAN [2]. GORDIAN builds a prefix tree of the data in order to find all maximal non-uniques, from which it computes all minimal uniques. However, Gordian does not consider datasets that are continuously changing. One could extend GORDIAN to deal with dynamic datasets, but updating the prefix tree and computing minimal uniques from maximal non-uniques every time the input dataset changes are two major performance bottlenecks (as seen in our experiment results).

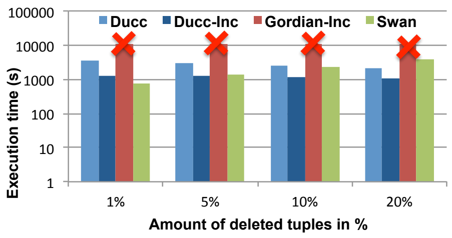
HCA is a column-based algorithm that performs an optimised candidate generation strategy, applies statistical pruning, and considers functional dependencies (FDs) inferred on the fly [5]. Recently, we proposed DUCC, a scalable unique discovery approach that, in contrast to Gordian and HCA, mainly depends on the solution set size [6]. Similarly to GORDIAN, HCA and DUCC consider only fixed-size datasets and, in contrast to GORDIAN, they have no optimization with regard to early identification of non-uniques.



(a) NCVoter with 5 millions rows and 40 columns



(b) Uniprot with 400k rows and 40 columns



(c) TPC-H with 5 millions rows and 16 columns

Fig. 7. Scaling the Number of deleted tuples

A related line of research to the unique discovery problem is discovering FDs in a given relation [4], [12], [13]. One of the best-known methods for FD discovery is TANE [4], which is a *levelwise*-based algorithm [14]. However, TANE works well only when the number of attributes is small. Other FD discovery algorithms [15] also follow a similar levelwise approach and hence they also may take exponential time in the number of attributes. FastFD [12], [16] improves these previous algorithms when the number of attributes is large, but it is more sensitive to the size of the input dataset. Some researchers, in fact, have incorporated the knowledge on existing FDs in order to identify those attributes that are, or are not, definitely part of uniques [17]. Another related topic to unique discovery is the discovery of conditional functional dependencies (CFDs) [18], [19], inclusion dependencies (INDs) [20], [21], and conditional inclusion dependencies (CINDs) [20], [22]. However, similar to unique discovery algorithms, none of these techniques consider dynamic datasets.

It is worth noting that most commercial relational DBMS allow users to specify a set of integrity constraints (such as uniqueness) over relations. The DBMS validates all the user-defined constraints after each inserted tuple and aborts an insertion in case it does not satisfy one of these constraints. However, the DBMS cannot find new uniques and non-uniques after a set of inserted tuples.

In summary, this paper is the first to address the unique discovery problem on dynamic datasets.

VII. Conclusion

We focused on the problem of finding all uniques and non-uniques on datasets that are continuously changing. Discovering all uniques and non-uniques in this context is cumbersome, because unique discovery is NP-hard in the number of columns and sub-quadratic in the number of rows. We presented SWAN, the first system to discover unique and non-unique constraints on dynamic datasets. SWAN is the first approach to mainly depend on the incremental data size ignoring the size of the initial dataset. SWAN makes use of intelligently chosen indices to minimize access to old data and speed-up the whole unique discovery process. We evaluated SWAN through exhaustive experiments. The experimental results show the high superiority of SWAN, *more than one order of magnitude faster* than the two state-of-the-art techniques GORDIAN and DUCC. In particular, the results show that SWAN even improves these two systems on the static case by dividing the dataset into a static part and a set of inserts.

References

[1] F. Naumann, "Data profiling revisited," *SIGMOD Record*, vol. 42, no. 4, 2013.

[2] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald, "Gordian: Efficient and Scalable Discovery of Composite Keys," in *VLDB*, 2006, pp. 691–702.

[3] Z. Lacroix and T. Critchlow, *Bioinformatics: Managing Scientific Data*, ser. The Morgan Kaufmann Series in Multimedia Information and Systems, 2003.

[4] Y. Huhtala, J. Kaerkkaeinen, P. Porkka, and H. Toivonen, "TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies," *The Computer Journal*, vol. 42(2), pp. 100–111, 1999.

[5] Z. Abedjan and F. Naumann, "Advancing the Discovery of Unique Column Combinations," in *CIKM*, 2011, pp. 1565–1570.

[6] A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann, "Scalable Discovery of Unique Column Combinations," *PVLDB*, vol. 7, no. 4, 2013.

[7] C. L. Lucchesi and S. L. Osborn, "Candidate keys for relations," *Journal of Computer and System Sciences*, vol. 17, no. 2, pp. 270 – 279, 1978.

[8] D. Gunopulos, R. Khardon, H. Mannila, and R. S. Sharma, "Discovering All Most Specific Sentences," *TODS*, vol. 28, pp. 140–174, 2003.

[9] Y. Huhtala, J. Kaerkkaeinen, P. Porkka, and H. Toivonen, "Efficient Discovery of Functional and Approximate Dependencies Using Partitions," in *ICDT*, 1998, pp. 392–401.

[10] J. Bauckmann, Z. Abedjan, U. Leser, H. Müller, and F. Naumann, "Discovering Conditional Inclusion Dependencies," in *CIKM*, 2012.

[11] C. Giannella and C. Wyss, "Finding Minimal Keys in a Relation Instance," http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7086, 1999, last accessed on 2013-02-21.

[12] C. M. Wyss, C. Giannella, and E. L. Robertson, "FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances," in *DaWaK*, 2001, pp. 101–110.

[13] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga, "CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies," in *SIGMOD*, 2004, pp. 647–658.

[14] H. Mannila and H. Toivonen, "Levelwise Search and Borders of Theories in Knowledge Discovery," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 241–258, 1997.

[15] T. Calders, R. T. Ng, and J. Wijsen, "Searching for Dependencies at Multiple Abstraction Levels," *TODS*, vol. 27, no. 3, pp. 229–260, 2002.

[16] S. Lopes, J.-M. Petit, and L. Lakhal, "Efficient Discovery of Functional Dependencies and Armstrong Relations," in *EDBT*, 2000, pp. 350–364.

[17] H. Saiedian and T. Spencer, "An Efficient Algorithm to Compute the Candidate Keys of a Relational Database Schema," *Comput. J.*, vol. 39, no. 2, pp. 124–132, 1996.

[18] W. Fan, F. Geerts, J. Li, and M. Xiong, "Discovering Conditional Functional Dependencies," *TKDE*, vol. 23, no. 5, pp. 683–698, 2011.

[19] L. Golab, H. J. Karloff, F. Korn, D. Srivastava, and B. Yu, "On Generating Near-Optimal Tableaux for Conditional Functional Dependencies," *PVLDB*, vol. 1, no. 1, pp. 376–390, 2008.

[20] F. D. Marchi, S. Lopes, and J.-M. Petit, "Unary and n-Ary Inclusion Dependency Discovery in Relational Databases," *Journal of Intelligent Information Systems*, vol. 32, no. 1, pp. 53–73, 2009.

[21] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava, "On Multi-Column Foreign Key Discovery," *PVLDB*, vol. 3, no. 1, pp. 805–814, 2010.

[22] L. Bravo, W. Fan, and S. Ma, "Extending Dependencies with Conditions," in *VLDB*, 2007, pp. 243–254.