Service Oriented Computing and Applications (2021) 15:323–338
https://doi.org/10.1007/s11761-020-00313-x

ORIGINAL RESEARCH PAPER

A third-party replication service for dynamic hidden databases

Stefan Hintzen[1] · **Yves Liesy**[2] · **Christian Zirpins**[3]

Received: 21 August 2020 / Revised: 2 December 2020 / Accepted: 9 December 2020 / Published online: 8 January 2021

Abstract
Much data on the web is available in hidden databases. Users browse their contents by sending search queries to form-based interfaces or APIs. Yet, hidden databases just return the top-k result entries and limit the number of queries per time interval. Such access restrictions constrict those tasks that require many/specific queries or need to access many/all data entries. For a temporary solution, an unrestricted local snapshot can be created by crawling the hidden database. Yet, keeping the snapshot permanently consistent is challenging due to the access restrictions of its origin. In this paper, we propose a replication approach providing permanent unrestricted access to the local copy of a hidden database with dynamic changes. To this end, we present an algorithm to effectively crawl hidden databases that outperforms the state of the art. Furthermore, we propose a new way to continuously control the consistency of the replicated database in an efficient manner. We also introduce the cloud-based architecture of a replication service for hidden databases. We show the effectiveness of the approach through a variety of reproducible experimental evaluations.

Keywords Dynamic hidden databases · Crawling · Replication-as-a-service

1 Introduction

A broad variety of providers are offering contents to the public by means of *hidden databases*. Hidden databases are online-accessible sources of data that provide a public interface for the controlled utilization of underlying database systems. They are widely established as *web databases* offering form-based interfaces or web APIs. Other types include public LDAP services.

Beyond different interface technologies, hidden databases impose significant restrictions on the utilization of their underlying database systems. They generally limit the means of data acquisition to Boolean *search queries* matching some (ranges of) attribute values [14]. Also, hidden databases follow a ranked retrieval model. They do not return all but a subset of *top-k* results to the client following some proprietary ranking function [7]. Moreover, hidden databases impose strict *query rate restrictions* [12]. In doing so, they limit the number of queries that each user can issue per time period to a given maximum.

Hidden databases aim for fast and compact results when interactively browsing a specific subset of data entries. However, there are cases where unrestricted access to a large part or even the full range of all entries is required. An example would be the indexing of all available contents in a web database by a crawler [18]. In other situations the re-ranking of query results is desired that requires to apply an individual ranking function to the full set of all relevant entries [3,7]. Then again, a local copy of all data entries can be used to implement data analysis by means of any possible operation without external restrictions [12]. Finally, it can be beneficial to maintain a client-side cache of entries for the purpose of serving queries locally and reducing expensive requests to the hidden database.

The problem of gathering all entries from a hidden database is known as *crawling*. Crawling all n entries of a hidden database is non-trivial because a naive query for *any entry* would just return the top-k results. Yet, querying *each entry* individually by enumeration would quickly hit the query rate limit. Algorithms like rank-shrink [18] create a set

✉ Christian Zirpins
christian.zirpins@hs-karlsruhe.de

Stefan Hintzen
stefan.hintzen@siemens.com

Yves Liesy
yves.liesy@siemens.com

[1] Siemens AG, Munich, Germany

[2] Siemens AG, Mannheim, Germany

[3] University of Applied Sciences, Karlsruhe, Germany

of optimized range queries covering all entries. They show a theoretical minimum cardinality of $O(d \cdot \frac{n}{k})$ for d-many numeric domains.

Crawling a hidden database for a current snapshot of entries is an essential step for their unrestricted access. For example, a web crawler might then create an index of all contents. Beyond that, we are especially interested in those cases, where unrestricted access is *permanently* required. For example, a client-side cache server must answer any query at any time with the same (but possibly more complete) results than the hidden database. However, by turning from momentary to permanent usage scenarios, we need to consider that the original contents may change. Entries might be inserted, updated or deleted in the hidden database at any time. In other words, we are concerned with *dynamic hidden databases*. In such cases, a reasonable solution for permanent unlimited access to the entries must include means for their maintenance including the management of updates.

Generally, maintaining the same data on multiple locations for improved quality of access while ensuring the consistency of the copies is known as replication. From that perspective, we are looking for a way of *replicating dynamic hidden databases*. Here, the autonomous nature of hidden databases leads to a limitation of possible replication methods. With no means to change the functionality of a hidden database itself, all we can do is to establish a client-initiated passive replication strategy with pull-based updates (aka *cache*). Clients must be caching entries individually and need to pull updates by means of re-crawling entries of the hidden database. This situation introduces various challenges:

1. *Initiating the cache* requires an efficient method for crawling the hidden database. While the theoretic cost of this task is known [18], there is still room for improved algorithms to increase performance.
2. A naive approach for *updating the cache* by continuous re-crawling of all data entries would lead to an infeasible amount of update queries. The high effort of such updates calls for a consistency model that allows to relax the level of consistency in order to optimize the amount of update queries.
3. The cost of *operating a cache* might be prohibitive for individual clients. Thus a practical software system must be designed that allows for adequate utilization of resources. Usually this involves a service model enabling to share between multiple clients.

To concretize the underlying problem, we assume a hidden database to offer a set of records U. Entries conform to a given schema with at least one continuous domain uniquely identifying any entry. We further expect that programmatic access of U is given via a query function Q that is part of a public API. For a conjunctive range query A, $Q(A)$ yields an effective result $E \subseteq U$, but returns a pseudo-random subset $E_g \subseteq E$ of maximum size g. The query rate is restricted and any two consecutive calls $Q(A)$ of the same query may yield different results due to the dynamics of U.

A hidden database cache is a local copy U_{loc} of U that is once initiated and continuously maintained by means of remote queries to Q with restricted results. Its local query function Q_{loc} is identical to Q. For any query, we require the result sets for $Q_{loc}(A)$ and $Q(A)$ to guarantee a given level of consistency at all time. In our work, we use *deviation of staleness* [22] as the measure of consistency. That is, query results from cached entries are guaranteed to show a maximum update delay.

The problem is then to find a replication method for the hidden database cache U_{loc} that allows to specify and maintain a given level of consistency while causing a minimum number of hidden database queries.

In this paper, we propose a general holistic solution for the problem of replicating dynamic hidden databases including two complementary methods of replication management and a cloud-based implementation. The first method (TRENCH) crawls all entries of a hidden database in order to create the initial contents of a cache replica. The second method (MINCORE) manages continuous updates of these cache contents in order to maintain a required level of consistency. A cloud-based system architecture adopts TRENCH and MINCORE in order to provide hidden database replication as a third-party service. More concretely, our contributions are as follows:

1. We present the TRENCH algorithm for efficiently crawling arbitrary ranges of a hidden database. The algorithm dynamically adjusts query ranges to the varying density of data entities with respect to the domains of U. Furthermore, TRENCH can effectively deal with subsets of similar values that exceed the top-k results and interruptions by query rate restrictions.
2. We introduce the MINCORE approach to plan and control optimized update strategies for a hidden database cache. It computes the effective minimal set of range queries for full cache updates. Yet, it adopts a client-centric, continuous consistency model controlling the staleness of the cache. We guarantee that a user will never see query results representing an older state of the hidden database than before a fixed timespan t. For this consistency model, MINCORE just requires partial cache updates.
3. We propose a cloud-based architecture for providing hidden database replication as a service. We show how the fundamental algorithms of TRENCH and MINCORE can be efficiently operated on top of the Amazon cloud platform. A prototype implementation is utilized

for extensive experimental evaluation of our approach including lab and field studies.[1]

We have shown the performance of TRENCH to surpass state-of-the-art solutions by around 10% on average. Furthermore, MINCORE clearly outperforms the baseline of naive updating as we have shown for search-based access patterns. All results are based on lab studies using realistic datasets. Beyond that, we have conducted a field study with a real world public LDAP service underpinning practical feasibility of our approach.

Our results suggest that the improvements of crawling with TRENCH as well as the novel update approach of MINCORE provide contributions that can clearly improve the utilization of hidden databases. With a growing number of such data sources at literally everybody's fingertips, these contributions can be expected to be significant in various practical cases.

In the following, Sect. 2 discusses related work. Section 3 prepares some definitions and terminology. Section 4 introduces our fundamental algorithms for crawling a hidden database to gather its entries. Section 5 proposes data consistency strategies and related mechanisms. Section 6 outlines the architecture and implementation of a cloud-based replication service. Section 7 presents a variety of evaluations that we have conducted with the replication system. Finally, Sect. 8 concludes this paper.

2 Related work

Most of the work that is relevant to hidden databases has been conducted on the broader field of deep web data extraction. Some focus on the identification of access points to deep web databases [1,4,10,13,23], like web forms and APIs. Others study valid call patterns for those databases [5,8,12,15]. All of them underpin the problem context of this paper, since the access point and domains of a hidden database have to be known to apply our approach.

The extraction of data through incomplete queries is another topic that has been addressed. A lot of work deals with queries that are incomplete with respect to missing attribute values in the entries of the (otherwise complete) result set [2,9,14,17,19,21]. A common solution to resolve this kind of incompleteness, as well as fuzzy searches over hidden databases in general, builds on query refactoring [14,17,21] and educated guessing of values [2,9,19]. However, such approaches are not usable in our problem context, since our notion of incomplete queries relates to restrictions of the result set size. Research focussing such incomplete queries is mostly addressing topics of aggregate functions and data sampling [11,16,20], as well as query re-ranking [3,7].

Algorithms proposed for query re-ranking might be applicable for hidden database crawlers. However, to the best of our knowledge, [18] is the only work proposing an algorithm to completely crawl a hidden database over ranged domains. Therefore, we consider its rank-shrink algorithm as state of the art for our use case and compare our crawling solution against it. Also, to the best of our knowledge, there has not been any research regarding the most performant way to re-crawl an existing hidden database copy for updates as we do.

3 Definitions and terminology

In the following section, we introduce the terminology and basic definitions used in this paper. The problem revolves around a finite set of data entries $d_1, ..., d_i$ stored in a foreign database U that we want to gather and re-offer by a replication service. Each entry consists of one or multiple values, where each value relates to one of a finite set of domains referred as *namespaces*. In this paper, we do not assume that each entry has a value for every namespace. However, we expect that there is at least one *universal namespace* that contains a unique identifier for each entry. Namespaces might be discrete or continuous domains, as long as lower bound and upper bound values can be specified.

To execute a query A against the database U, we use the query function $Q(A)$. An execution of A through $Q(A)$ will return a full result set E containing all entries matching the filter criteria specified in A. When executing a query A against a local database copy, we mark the respective query function as $Q_{loc}(A)$.

Entries in the database U are queryable over the different namespaces through an API, which we will call the *foreign API*. The foreign API is defined by its behavior, where a query A is restricted to a result set E_g with maximum cardinality g. Hereby, g is a fixed value specified by the foreign API. When the complete result would contain more than g-many entries, the foreign API will return a pseudorandom subset of g-many distinct entries from the full result set. We define this behavior by means of a restriction function over the original query

$$B_g(Q(A)) = \begin{cases} |E| \leq g : & E \\ |E| > g : & R_g(E) \end{cases}$$

$$R_g(E) = \textit{pseudorandom subset of } E \textit{ with cardinality } g$$

Each query A consists of at least one range filter $a = \langle a_v, a_{start}, a_{end} \rangle$. Hereby, a_{start} and a_{end} define the range

[1] We provide all source code and datasets on Github at https://github.com/HSKA-IWI-VSYS/hd-cache.git.

$[a_{start}, a_{end})$ of the filter a, while a_v defines its *volume dimension*. The term volume dimension refers to the view of an entry or a set of entries restricted to the values of one namespace. For example, the volume dimension "last name" related to a telephone book is a view restricted to the last name of each entry ignoring all other namespaces such as "first name" or "address". A volume dimension only contains entries that provide a value for the specified namespace. Entries without a value for the namespace are ignored. A data entry d or an entry set E might be projected to a volume dimension v by applying the volume dimension function V as $V_v(d)$ and $V_v(E)$. We define $N_v = V_v(U)$ as the set of all existing values in the namespace viewed through v.

We check whether an entry d resides in the range and volume dimension of a filter a using the filter function

$$F(a, d) = a_{start} \leq V_{a_v}(d) < a_{end}$$

Subsequently, we define the execution of a query A with filters $(a_1, ..., a_n)$ against the database U as

$$Q(A) = \{d \in U | \bigwedge_{i=1}^{n} F(a_i, d)\}, A = (a_1, ..., a_n)$$

This formula also applies for executing A against a local database copy through $Q_{loc}(A)$, where U is replaced by U_{loc}. We refer to the desired state of the local database copy U_{loc} as *volume consistency*. Volume consistency on a range $[x_{start}, x_{end})$ for a volume dimension v means that U_{loc} can fully serve any query with a filter $a = \langle v, x_i, x_j \rangle$. Here, $[x_i, x_j)$ is a sub-range of $[x_{start}, x_{end})$. We define volume consistency on a range $[x_{start}, x_{end})$ for a volume dimension v as

$$\forall d \in U : (x_{start} \leq V_v(d) < x_{end} \rightarrow d \in U_{loc})$$

Every range query with a sufficiently small result set in order not to get restricted by the foreign API, will return a result set that contains every entry in its specified range. Therefore, we call the result of a query A to be volume consistent if $Q(A) = B_g(Q(A))$. Note that a volume consistent range is bound to one volume dimension. As a result, queries $A = (a_1, \ldots, a_n)$ with more than one filter cannot return volume consistent results. The reason is that a filter a_i might exclude entries that match another filter a_j and would be required for volume consistency on the range specified by a_j.

In contrast, *volume inconsistency* means that not every query on the specified range and volume dimension can be answered completely from the local copy U_{loc}. Note that volume consistency and inconsistency are anti-equivalent to each other.

4 Data acquisition

In this section, we will explain how to gather the complete amount of data entries by crawling a foreign API with the minimal amount of queries.

First, in Sect. 4.1, we introduce the algorithm behind the data extraction process. Next, in Sect. 4.2, we show how to extend the algorithm to handle parts of namespaces that cannot be crawled reliably with a single volume dimension. Finally, Sect.in 4.3, we show how to prepare gathered data for further update operations.

4.1 Extracting hidden data

To locally answer queries for the foreign API, we need to make sure that the local system returns at least the same results. Moreover, we aim to return *all* matching entries, not only a limited subset. This requires access to all data entries stored in the database U behind the foreign API, which is only achievable by creating and maintaining a consistent copy U_{loc} of U. This is a challenge as we neither know how many entries exist in the hidden database, nor how they are distributed across the namespace. Moreover, responses for our queries to the foreign API are restricted to g-many entries.

For the task of gathering all entries with the minimal amount of queries, we introduce the ***TRENCH*** algorithm. It aims to issue range queries with volume consistent results against the foreign API. This is done in such a way that the combined ranges completely cover a whole namespace from its lower to upper bound values. Thus it is guaranteed that every existing entry in the namespace is included.

To minimize the number of queries sent to the foreign API, TRENCH modifies the search range of subsequent queries. This optimization depends on the results of prior queries as well as the already found entries stored in U_{loc}. Since TRENCH is also applied to update entries (Sect. 5), we describe the algorithm in a more universal context. Here, it can gather data for any range $[x_{start}, x_{end})$ of a namespace. We will refer to the special case of gathering the full range as TRENCH_{full}. Algorithm 1 shows the pseudocode.

Algorithm 1 TRENCH-Algorithm

Require:

x_{start} ▷ Search range lower bound
x_{end} ▷ Search range upper bound
U_{loc} ▷ Partial copy of foreign database U
$step$ ▷ Step width depending on data distribution
v ▷ Volume dimension to consider
g ▷ Maximal result size for the foreign API
$Q(A)$ ▷ Full set of result entries from U for query A
$Q_{loc}(A)$ ▷ Set of result entries from U_{loc} for query A
$B_g(E)$ ▷ Restricted entry set with cardinality g
$V_v(E)$ ▷ All values in set E on volume dimension v

1: $step_{orig} \leftarrow step$
2: **while** $x_{start} < x_{end}$ **do**
3: $a_{saved} \leftarrow \langle v, x_{start}, x_{end} \rangle$
4: $x_{next} \leftarrow SCAN(V_v(Q_{loc}(a_{saved})))$
5: $x_{start} \leftarrow EXTRACT(x_{next})$
6: **end while**
7: **procedure** SCAN(E_{loc})
8: $res_{loc} \leftarrow$ *top-*$(g+1)$ E_{loc} *entries in ascending order*
9: **if** $step_{orig} \leq |res_{loc}|$ **then**
10: $step \leftarrow step_{orig} - 1$
11: **while** $res_{loc}[step] = x_{start}$ **do**
12: $step \leftarrow step + 1$
13: **end while**
14: **assert** $step \leq g$ ▷ LODIS case (see 4.2)
15: **return** $res_{loc}[step]$
16: **else**
17: **return** x_{end}
18: **end if**
19: **end procedure**
20: **procedure** EXTRACT($x_{current}$)
21: $a \leftarrow \langle v, x_{start}, x_{current} \rangle$
22: $U_{loc} \leftarrow U_{loc} \cup B_g(Q(a))$
23: **if** $B_g(Q(a)) \neq$ *volume consistent* **then**
24: $x_{next} \leftarrow SCAN(Q_{loc}(a))$
25: **return** $EXTRACT(x_{next})$
26: **else**
27: **return** $x_{current}$
28: **end if**
29: **end procedure**

TRENCH consists of multiple consecutive rounds of $SCAN$-ning local results (lines 7-19) and subsequently $EXTRACT$-ing additional entries from the foreign database (lines 20-29). The $SCAN$-procedure prepares a sample res_{loc} of the current top-$(g+1)$ local result entries containing values on the volume dimension v in the search range $[x_{start}, x_{end})$. It also identifies the leftmost sub-range $[x_{start}, res_{loc}[step])$ of a minimum size $step$ within this sample that may vary in width based on the density of entries in the given range. The $EXTRACT$-procedure then probes the sub-range for value consistency by querying the same range from the foreign API. res_{loc} is sorted by ascending order and limited to the first $g+1$ values, since a range $[x_{start}, res_{loc}[step])$ with $step \geq g+1$ would contain at least $g+1$-many entries forcing a volume inconsistent result.

A $step$ width of $g/2$ results in low numbers of queries against the foreign API for searching ranges completely. This value showed to be optimal for a uniform distribution of values in the namespace independent from the foreign API's limit g and the search range.

When the query for $[x_{start}, res_{loc}[step])$ returns a volume consistent result, the algorithm proceeds to the next round with $res_{loc}[step]$ as new lower search bound, thus replacing the old value of x_{start}. This behavior repeats until the condition $x_{end} \leq x_{start}$ is met. When this case applies, TRENCH has finally reached the upper search border and terminates.

We also enter a new round on a volume inconsistent result, but keep x_{start} unchanged. However, we let the $SCAN$ procedure extend the current sample res_{loc} with the values of the volume inconsistent result and proceed on that. This way, we inject enough new values into res_{loc} to make it impossible for $res_{loc}[step]$ to keep the same value. Moreover, since the upper border of a range is excluded in a query filter, all injected values have to be smaller than the old value of $res_{loc}[step]$. Thus, we can guarantee that the value of the upper border as well as the search range decrease for each new round triggered by a volume inconsistent result. It thus converges towards the smallest possible search range of just one value that will have to return a volume consistent result and thus advance the lower search border.

For TRENCH$_{full}$ that gathers all data entries in U, we simply run the algorithm on the range $[x_{min}, x_{max})$, where x_{min} is the lower bound of the namespace and x_{max} its theoretical upper bound. This range covers every possible value and thus every data entry in U.

TRENCH can be run in one or multiple rounds. This is necessary when dealing with hidden databases, since foreign APIs block requests after a specific maximum amount of requests have been answered in a given timeframe. In that case, it is necessary to wait until the next timeframe begins before the foreign API starts to answer requests again. TRENCH can work under such conditions, since the algorithm can be paused after every round. Thus, after saving the current search bounds, it is possible to resume the algorithm anytime by restarting it on the remaining search range.

4.2 Handling indivisible ranges

To avoid TRENCH querying empty ranges, we increase $step$ until the interval $[x_{start}, res_{loc}[step])$ does not show the same value for upper and lower bounds (lines 11-13). However, if res_{loc} already contains more than g-many values equal to x_{start}, there is no value for $step$ with $res_{loc}[step] \neq x_{start}$. Here, the smallest non-empty range $[x_{start}, x_{start}]$ on the searched volume v would yield a volume inconsistent result. Thus, there is no way for TRENCH to search a range volume consistently if it contains a value shared by more than g-many entries. Therefore, we propose the ***LODIS*** algorithm extending TRENCH for *Lower Dimension Search*.

When we recognize the need for lower dimension search in TRENCH (violation of assertion in line 14), we pause exe-

cution and remember the current value of x_{start} as x_p. Then we start a separate run of ***L-TRENCH*** on a different volume dimension v_{unique} that provides a view over a namespace with unique values for all data elements. Note that such a volume dimension is always guaranteed to enable a partitioning of volume consistent search ranges over any set of elements including those with value x_p on volume dimension v.

L-TRENCH is nearly identical to TRENCH, with the only difference that any range query A that would be executed to crawl volume dimension v_{unique} is substituted by $A_{lodis} = A \cup a_{lodis}$. A_{lodis} extends A by an additional filter $a_{lodis} = \langle v, x_p, x_{psucc} \rangle$ representing range $[x_p, x_p]$ with x_{psucc} being the successor of value x_p on volume dimension v. This way, we will not crawl the whole namespace in L-TRENCH again, but only the range not queryable through TRENCH on its volume dimension v. We turn back to the former TRENCH once L-TRENCH reaches the upper bound value of the namespace viewed over v_{unique} and therefore terminates. To this end, we increase x_{start} to x_{psucc}, the next possible value on the namespace viewed through v, and resume the execution of TRENCH.

4.3 Planning for updates

Since we aim to return every entry of U on request, we initially run TRENCH once with a complete namespace as search range. For completeness, this namespace must contain a value for every entry in U. Entries sharing the same value are handled by TRENCH for up to g-many occurrences and by L-TRENCH otherwise. The result is a copy U_{loc} of the hidden database U.

This copy U_{loc} is a momentary snapshot of U. In order to maintain U_{loc} in a state that remains consistent with U, we need a strategy to update U_{loc} with a minimal amount of queries. To this end, we introduce ***MINCORE***, an update plan dividing the initially queried range into a set of disjunct sub-ranges called *splinters*.

We optimize each splinter's sub-range to query a maximum number of entries while keeping the result volume consistent. The set of splinters covering the full namespace represents the minimal number of queries against the foreign API that is necessary to update U_{loc}. Since we calculate splinters based on the initial snapshot of U, we need to anticipate interim changes. In particular, ranges of size g would lead to volume inconsistent queries for any added entry in the same range of U. Thus we create splinters of size $g - p$, where p is a buffer for possible new entries. The value of p should be chosen dependent on the expected update frequency of U. We will further study the impact of p on MINCORE's performance in 7.3.3.

The optimum splinter size of $g - p$ has to be further decreased for the special case that the upper bound of a splinter's range is a non-exclusive value shared by multiple entries on the same volume dimension v. Here, the range would cover all of those entries and may exceed the maximum number of $g - p$ result entries. In order to decrease the probability of a volume inconsistent query result, we introduce an acceptable error o_a for filter $a = \langle v, x_i, x_j \rangle$ representing range $[x_i, x_j)$. This reduces the optimum splinter size to $g - p - o_a = |Q_{loc}(a)|$ if $|Q_{loc}(a)| + |\{d \in U_{loc} | V_v(d) = x_j\}| > g - p$ else $o_a = 0$.

We refer to the final range that touches the upper bound x_{max} of the namespace as *head splinter*. The head splinter is also likely to fall below the optimum splinter size as it has to cover the remaining part. This also applies for areas crawled through LODIS, where the LODIS head splinter touches the upper bound value of the namespace viewed through v_{unique}.

Note that a MINCORE M_v is only applicable for updating U_{loc} over one volume dimension v. However, we can minimize the number of queries to the foreign API by updating U_{loc} through a variety of volume dimensions. Therefore, we build one MINCORE for every queryable volume dimension. Thus, we can handle updates on all volume dimensions queried by users.

5 Data maintenance

We have discussed update concepts in Sect. 4.3 and presented MINCORE as an update plan. Next, Sect. 5.1 shows how to prepare the hidden database cache for our proposed update process. In Sect. 5.2, we explain how to use MINCORE in order to realize a client-centric model to keep the cache U_{loc} consistent with U.

5.1 Update preparations

Entries in U might change during or after the initial execution of TRENCH, which would lead to U_{loc} not being consistent to U anymore. Since full consistency between U and U_{loc} in a query's range is required in order to serve the correct entries, we will now show how to use MINCORE to capture relevant changes and transfer them into U_{loc}.

A splinter of a MINCORE M_v is a log of the last time, when a range $[s_{start}, s_{end})$ was queried on volume dimension s_v. We define a splinter s as a 6-tuple $s = \langle s_v, s_{start}, s_{end}, s_{ts}, s_{startL}, s_{endL} \rangle$ representing a query on s_v for range $[s_{start}, s_{end})$ that was run last on timestamp s_{ts}. If a range was crawled by LODIS (because s_{start} existed in more than g-many entries in U), then we narrow the search on volume dimension v_{unique} through the range $[s_{startL}, s_{endL})$ that is absent otherwise. For brevity, we refer to a splinter as *L-splinter* if s_{startL} and s_{endL} are present.

Also, we define *querying a splinter* s by $Q(s)$ as querying the corresponding filter $a_s = \langle s_v, s_{start}, s_{end} \rangle$ through

$Q(a_s)$ and *querying an L-splinter* s_l through $Q(s_l)$ as querying the corresponding filter set $A_L = \{\langle s_v, s_{start}, s_{end}\rangle, \langle v_{unique}, s_{startL}, s_{endL}\rangle\}$ by $Q(A_L)$.

A MINCORE M_v on volume dimension v is considered *complete*, if every possible value $x \in N_v$ is covered by the range of exactly one regular splinter s or a set of L-splinters $L \subset M_v$. If x is covered by L, then each value $y \in N_{v_{unique}}$ has to be covered by exactly one L-splinter $s_l \in L$. Formally, M_v is *complete* on v if $\forall x \in N_v, \forall y \in N_{v_{unique}}$, $\exists! s \in M_v : (s_{start} \leq x < s_{end} \wedge s_{startL} = \text{NIL})$ XOR $(s_{start} = x \wedge s_{startL} \leq y < s_{endL})$. We define a complete MINCORE $M_v = (s_1, ..., s_n, s_{head})$ without L-splinters to be *perfect* if $\forall i \in [1..n] : |Q_{loc}(s_i)| = g - p - o_{s_i} \wedge |Q_{loc}(s_{head})| \leq g - p$ is given.

Similar conditions apply if M_v covers some range crawled by LODIS, but then all head splinters of L-splinters might also be smaller than $g - p$. A perfect state corresponds to the optimal structure of a MINCORE since the update plan it represents covers the whole namespace with the smallest amount of queries.

Using MINCORE, we could periodically perform complete updates of U_{loc} by executing all queries specified by its splinters. However, any MINCORE M_v contains at least $|V_v(U)|/(g-p)$ splinters, which would lead to an unfeasibly large number of periodical queries.

Thus, we propose a demand-driven approach that utilizes real-time updates for limited sub-ranges of U_{loc} as requested by user queries. Ranges that are not requested will not be updated. In other words, we follow a client-centric consistency model, where consistency is only guaranteed for those parts of U_{loc} that are visible to users, while others might be outdated. Assuming that user queries hit popular ranges more often, this decreases the number of update queries significantly.

5.2 Integrated update method

Before serving a request to U_{loc} for some query A_{req}, we utilize MINCORE to run corresponding updates. In particular, we identify the filter $a_{fresh} \in A_{req}$ that intersects with the smallest amount of outdated splinters. The combined ranges of these splinters constitute the update queries. It is unnecessary to query intersecting splinters of any other filter from A_{req}, since these just narrow down the requested range. Thus, updating data entries in the range of a_{fresh} covers the complete query A_{req}. A splinter s is considered *outdated* when a fixed timespan t has passed since s_{ts}. Choosing t is subject to customization, where smaller t lead to more queries and higher consistency of U_{loc}.

We do not obtain outdated splinters directly from the splinter database table, but a separate *maintenance list*. Outdated splinters get transferred into the maintenance list on a regular basis, e.g., daily. Utilizing the maintenance list allows managing the ranges due for updates without manipulating the MINCORE itself. As a result, we can delete an outdated splinter from the maintenance list as soon as an update over its range has started. In practice, this prevents an outdated splinter from wrongly triggering an update while another one is still processing on the same range.

Outdated splinters represent a past state of U_{loc} older than t. Thus, their ranges might no more cover the optimal number of entries because U_{loc} changed. Therefore, we adjust outdated splinters $(s_1, \ldots, s_n)$ to the current state of U_{loc} before we query the foreign API for updates. First, we check whether any ranges of outdated splinters $(s_i, \ldots, s_j)$ from $(s_1, \ldots, s_n)$ are neighbors on the same volume dimension. We combine neighboring ranges into $[min_{s_{start}}(s_i, \ldots, s_j), \ max_{s_{end}}(s_i, \ldots, s_j))$. Function $max_x(s_i, \ldots, s_j)$ returns the biggest value and function $min_x(s_i, \ldots, s_j)$ the smallest value for element x of tuples $(s_i, \ldots, s_j)$. For L-splinters we use s_{startL} and s_{endL} instead, but only combine the ranges of two L-splinters if they share the same value for s_{start}.

Next, we create a new set of perfect MINCOREs for the combined ranges and all remaining ranges of preselected outdated splinters. The newly created splinters, referred as *navigators*, represent the optimal update plan reduced to outdated parts of user-requested ranges. Formally, a navigator z is a 5-tuple $\langle z_v, z_{start}, z_{end}, z_{startL}, z_{endL}\rangle$. Thus, navigators are similar to splinters, except for the missing timestamp, and can be queried through the query function $Q(z)$.

Note that each of the newly created MINCOREs contains a head splinter that is likely to cover a suboptimal number of entries. Hence we extend the upper range bounds of head navigators z_i in such a way that each one covers exactly $g - p - o_{z_i}$ entries. Such head navigators might reach into parts of the namespace that are neither due for an update, nor requested by the user. However, since a query is required anyway, we can use it to update as much of the namespace as possible. Yet, if the head navigator overlaps with the navigator of another newly created MINCORE, it is better to recalculate a single MINCORE for the combined ranges.

Each extended head navigator z_i also intersects a splinter s_{i+j} with $j \geq 1$ on the same volume dimension. Hence, we adapt the lower range bound of s_{i+j} to the value of z_i's upper range bound after extending z_i to optimal size. Splinters completely covered by z_i are obsolete, since the update specified by them is already covered by querying z_i. Therefore, we delete them from the splinter table and the maintenance list to prevent simultaneous updates over the same range.

After calculating all navigators, we send respective queries to the foreign API and check each result set for volume consistency. If the result of some query for navigator z_i is volume consistent, we substitute the entries in the respective range of U_{loc} with the result set. If the result set is volume inconsistent, then the gap between U and U_{loc} became too big to calculate

an update plan from U_{loc}. Here, we resort to TRENCH to completely search the range and reestablish volume consistency. If multiple neighboring ranges need to be searched by TRENCH, we combine them into one range before executing the algorithm. Once all updates got processed, U_{loc} is volume consistent again in the range of A_{req} and the local result set can be returned to the client.

The final step is to adapt the MINCORE M_v to represent the updated state of U_{loc}. We do this by deleting all splinters intersecting the range of a navigator that got queried on the volume dimension v. This leads to a number of volume consistent ranges, referred to as *holes*, that are not covered by any splinters. To fix a hole in M_v, we calculate a perfect MINCORE M_{temp} in the hole's range on v and merge M_{temp} into M_v. This way, we cover the hole in M_v with splinters from M_{temp}.

A MINCORE is complete again once the holes are fixed. Yet, fixing a hole might make it lose its perfect state. Fortunately, MINCOREs will regain perfect state over time as calculating M_{temp} merges all suboptimal-sized splinters in its range into optimal-sized splinters and only one suboptimal-sized head splinter.

6 Hidden database replication service

From the concepts described so far, we have designed a data replication service for hidden databases and a prototype of this replication service has been implemented for the Amazon Web Services (AWS) platform.

In the following, Sect. 6.1 first outlines the service model for hidden database replication and then explains its architectural design. Sect. 6.2 describes the AWS-based services used to implement a prototype.

6.1 Service model and architectural overview

Our approach is meant to improve the access to hidden databases regarding the type of queries clients can issue and the amount of results they receive in return. We translate this into the model of a cloud-based *Hidden Database Replication Service (HDRS)* as follows.

A HDRS relates to a given foreign database U for which it maintains a single local database copy U_{loc}. All clients of the HDRS share the same instance of U_{loc} and access it through $Q_{loc}(A)$ by an API. In this paper, we confine $Q_{loc}(A)$ to be identical to $Q(A)$ but it could choose to offer extended query functions. Note that no data isolation is required as all tenants access identical public data. On the contrary, queries from different tenants help improving consistency for each other.

As described before, there are three main tasks that the HDRS has to handle. This includes 1) serving user requests, 2) updating U_{loc} with retransmitted data from U and 3) optimizing the MINCOREs to schedule future updates. These tasks translate into two services of an architectural design that is shown in Fig. 1.

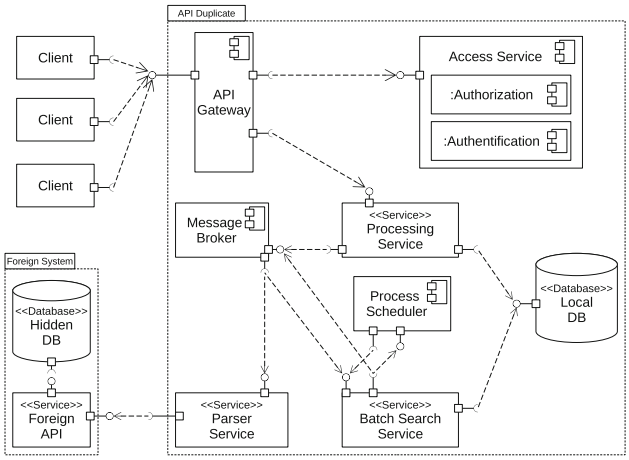

Fig. 1 Architecture of the HDRS in UML-notation

Generally, update control is tightly coupled to serving user requests and thus combined into the *Processing Service (PS)*. The *Batch Search Service (BSS)* retrieves ranges in a volume consistent way through TRENCH and handles manipulations of MINCOREs.

Before starting to accept user requests, the system executes an initial run of TRENCH in the BSS to extract all data from U. This functionality is not exposed to the API but only triggered internally.

For any user request A, the PS executes update queries as deducted from the outdated splinters in the requested range. Then, it adjusts the maintenance list and splinter table and queries U_{loc} for all existing entries fitting the filters of A. This way, all tasks necessary for serving requests are bundled in one service process thus avoiding request forwarding. Since the PS aims to answer requests as fast as possible, it delegates time-consuming tasks to the BSS. Those tasks include execution of TRENCH on given ranges and subsequent recalculation of the respective MINCORE.

PS and BSS communicate asynchronously by messaging to realize non-blocking service requests. Optionally, the PS may issue blocking update requests to wait for the BSS finishing its TRENCH run. This way, user queries are synchronized with the update procedure and therefore guaranteed to return the latest results from U. However, TRENCH sends many requests to the foreign API synchronously, which may result in high latency. The default behavior of the PS is to instantly respond to user requests while TRENCH still executes.

When the PS or BSS need to query the foreign API, the request is forwarded to the *Parser Service*. Its purpose is to translate query requests into the format required by the foreign API and pass them on. This decouples generic

replication logic from a specific foreign API and bundles all dependent configurations and code. For further decoupling, we utilize a message broker that routes task messages between the services. The message broker needs to provide means for asynchronous as well as one way requests with QoS-guarantees.

Users issue query requests to the HDRS through an *API gateway*. It exposes a public REST-API and forwards user requests to the PS for further handling. Other services and components are encapsulated behind the API gateway. Furthermore, to secure the PS from unwanted access, e.g., through bots, the API gateway is connected to an *Access Service*. It handles the authentication and authorization of user requests. This way, we may block users or bots, e.g., if they send an unreasonable high amount of requests.

Handling of API Lockdowns So far the proposed architecture can replicate a hidden database, as long as the foreign API answers queries. However, there might be times when this is not possible. Hidden databases stop serving queries once they have answered an upper bound number of requests per time [1,11]. We refer to this state as a *lockdown*. In order to remain functional during API lockdown, we temporarily adapt behavior.

Data gathering through TRENCH is most likely to encounter lockdowns as it includes many queries to the foreign API in short time. Once a lockdown occurs, the HDRS needs to stop its TRENCH run and persist remaining ranges. Since TRENCH is pauseable, we restart it once the foreign API releases the lockdown. That is, an external process scheduler periodically triggers TRENCH on the remaining search range. Following this procedure, we can handle data extractions for ranges of all sizes, even if a lockdown occurs in between.

Unlike data gathering, we cannot postpone the execution of a user request when a foreign API enters lockdown state. Therefore, we answer the request with data from U_{loc} in such cases even if outdated splinters intersect the requested range. While this will lower the quality of results, it still returns the most up-to-date data available. When expecting lockdowns, it is also advisable to delete splinters only once a result set got returned for their queries. This way, we can prevent holes in a MINCORE that cannot be fixed due to the foreign API suddenly locking down.

6.2 Prototype implementation

We have implemented a prototype of the HDRS architecture to show its feasibility and effectiveness. It builds on the AWS PaaS-Cloud that provides platform services for most infrastructure-related tasks. In detail, we implemented the parts of the HDRS as follows:

API gateway. We use the AWS-service *API-Gateway* to expose a public REST-API as entry point.

Access service. Access to to the API-Gateway is controlled by the AWS Cognito service. A Cognito user pool is used to manage user credentials for accessing the API. Users might be banned from the HDRS by removing them from the user pool.

Process service, batch search service. These services run on the AWS Lambda platform as Function-as-a-Service (FaaS). Each service is represented by a separate stateless function based on Node.js. AWS Lambda routes requests to the respective functions and automatically launches their runtimes on demand. Moreover, it handles the scaling of services by distributing function calls over servers of selected data centers. As a result, the interactive parts of the HDRS run and scale on demand, which saves resources in the long run.

Message broker. The prototype does not use an explicit messaging service, as the AWS-SDK already provides routing functions. These include event-based requests for communication between PS and BSS.

Parser service. In terms of the parser service, we have also simplified the proposed architecture. The added modularity of a separate service seems favorable for production systems. For the prototype we have integrated the parser functionality into the PS and BSS directly.

Process scheduler. Rescheduling of TRENCH runs has not been implemented in the prototype but AWS CloudWatch could be used to implement this feature.

Local Database. The database for U_{loc}, splinter table and maintenance list builds on MariaDB that is provided by the AWS Relational Database Service (RDS).

All services operate inside an AWS Virtual Private Cloud. It provides a cluster inside a logically separated sub-area of the AWS cloud. This way, we prevent illegal access from external systems inside and outside of the AWS cloud. Communication between the AWS services is handled automatically by AWS. For calls to other AWS services inside the code of an AWS Lambda function, we utilize the AWS-SDK. The only exception is for RDS, where we connect to MariaDB directly by means of a native driver.

7 Evaluation

Following, we present an experimental evaluation of the proposed concepts. We have compared the performance of TRENCH and MINCORE against current state-of-the-art solutions under various conditions. Furthermore, we present a field study of the replication service in a real-world setting underpinning its feasibility.

Initially, Sect. 7.1 describes the general setup regarding storage and compute resources as well as datasets used

for experiments. Next, section 7.2 shows a comparison of TRENCH with *rank-shrink* [18], the current state-of-the-art algorithm for crawling hidden databases. Then, section 7.3 compares MINCORE with periodic rebuilds of U_{loc} via TRENCH$_{full}$. This is currently the most relevant alternative, because, to the best of our knowledge, updating a database over a hidden database API has not been subject to former research. Finally, sect. 7.4 shows the feasibility of our prototype by replicating parts of a public LDAP directory. This confirms the results gathered under laboratory conditions in a real-world context.

7.1 Experimental setup

We run all lab experiments in an environment equivalent to the cloud-based implementation described in 6.2 using a machine with six cores, 2,6 GHz Intel processor and 16GiB memory. The field study described in 7.4 was conducted on the cloud implementation itself. Here, we used a t2.small-instance of MariaDB for RDS with 1vCPU (3,3 GHz Intel processor) and 1GiB memory. AWS Lambda functions have been configured to use 1024 MB of RAM. We configured the database (tables) with the collation *utf16_bin*. Thereby, comparisons in node.js and the database yield the same results. Also, the system compares and sorts case-insensitive to support protocols like LDAP.

Confirming the correctness of evaluation results requires some ground truth. More concretely, we need information on size and content of U to compare it with U_{loc}. Unfortunately, this is not available in the context of a real foreign API, where U is hidden. Therefore, we evaluate the effectiveness of TRENCH and MINCORE against a local database U_{hd} emulating a foreign API. U_{hd} returns a pseudo-random selection of g-many entries when an unrestricted query would return a larger result set. The contents of U_{hd} build on two datasets that have been used for different experiments:

NAMES. A family of datasets have been derived from the most common surnames according to the U.S. Census from 2000. The *names* dataset[2] includes a total of 151671 surnames (*name* attribute) together with the numbers of U.S. households carrying each name (*count* attribute). We use *names* to derive multiple datasets NAMES$_x$ that vary in size while preserving the original frequency of surnames in relation to other surnames.

For a dataset NAMES$_x$, we create an amount $n_i = \lfloor names[i].count * C \rfloor$ of 2-tuples $(names[i].name, id_{i,j})$ for each $i \in (1, \ldots, x)$ and $j \in (1, \ldots, n_i)$. Each $id_{i,j}$ is a unique identifier and C is a constant factor with $C = 1/names[x].count$ that scales the occurrences of surnames thus controlling the dataset size.

LANDSLIDES. The LANDSLIDES dataset contains information on rainfall-triggered landslide events gathered by NASA in the *Global Landslide Catalog*. In contrast to NAMES, LANDSLIDES is not an artificial dataset and will be used as provided by NASA[3] to show that TRENCH and MINCORE are usable on real data from the public web. The original dataset contains a number of 11033 entries with 31 attributes (retrieved 12/31/2019). We have transformed them into 6-tuples of $(event\_id, event\_title, source\_name, event\_date, country\_name, landslide\_setting)$ where $event\_id$ is representing a unique identifier.

NCSU. Unlike former static datasets, NCSU is a public foreign LDAP API[4] It is operated by the *North Carolina State University (NCSU)* and provides public information on students, employees and the campus. The foreign API enforces a limit of $g = 500$ result entries.

We use NAMES and LANDSLIDES to run controlled experiments evaluating the performance of the TRENCH and MINCORE algorithms compared with state-of-the-art approaches. We measure individual performance by the amount of queries that need to be send to the foreign API for a predefined task, where less queries indicate better performance. This is because queries are responsible for the vast majority of time and resources required by the given problem class. We use NCSU for a field study to underpin the feasibility of our system. Here, we report on running times and data sizes experienced under real-world conditions.

Generally, we took measures to comply with privacy laws. We strictly used datasets from the public domain for $NAMES$ and $LANDSLIDES$ as well as the data offered through $NCSU$. We also studied the *Terms of Service* for $NCSU$ to make sure that it does not forbid automated queries as well as storing of the queried data. U_{loc} was set up on a secured database in a private AWS VPC, which ensured that the database was not publicly accessible. Furthermore, we also operated all other parts of our system in that VPC to secure them from unauthorized access. Finally, we disposed all personal data gathered through $NCSU$ as soon as it was no longer required for evaluation.

7.2 Evaluating TRENCH

To evaluate TRENCH, we have compared it with rank-shrink, the state-of-the-art for crawling hidden databases. In 7.2.1 we let both algorithms crawl several NAMES$_x$ of varying size $|U|$. In 7.2.2 we run the algorithms on LANDSLIDES with varying restrictions g.

[2] Get the Census dataset from https://www.census.gov/topics/population/genealogy/data/2000_surnames.html.

[3] Get LANDSLIDES from https://data.nasa.gov/Earth-Science/Global-Landslide-Catalog-Export/dd9e-wu2v.

[4] NCSU is queryable at ldap://ldap.ncsu.edu:389.

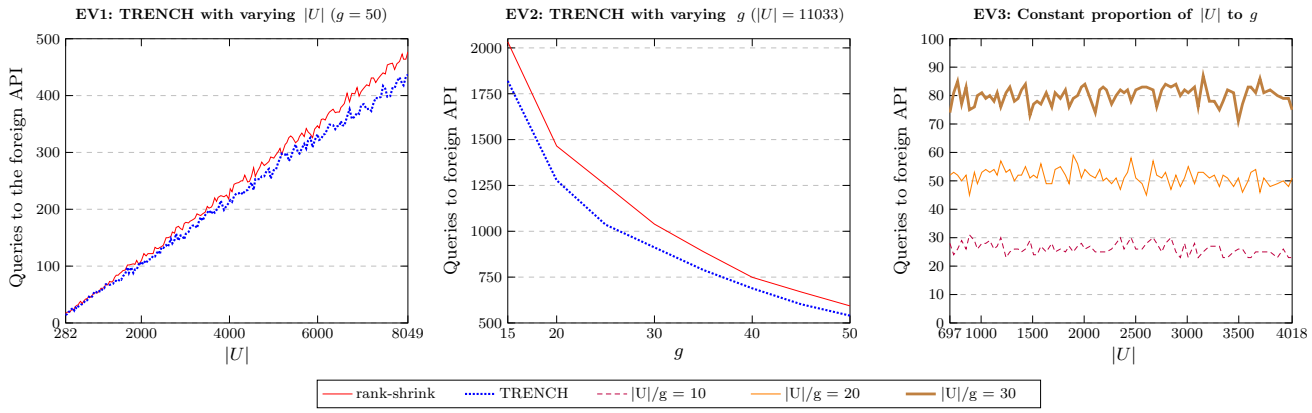

Fig. 2 Comparison of TRENCH against rank-shrink

7.2.1 NAMES$_x$ with varying $|U|$

First, we run TRENCH as well as rank-shrink against NAMES$_x$ with $x \in \{100, 110, 120, \ldots, 1790, 1800\}$ over the namespace *name*. This leads to 171 versions of U_{hd} with sizes ranging from $|U_{hd}| = 282$ for NAMES$_{100}$ to $|U_{hd}| = 8049$ for NAMES$_{1800}$. We set $g = 50$ as restriction regarding maximum results from U_{hd}.

The results are shown in Fig. 2 (EV1). Both algorithms exhibit fluctuating query performance revolving around a linearly growing median. Hereby, the median performance of TRENCH is about 10% better than for rank-shrink. For small database sizes rank-shrink can outperform TRENCH if the pseudo-random query results are causing the algorithms to run into opposite best-case/worst-case scenarios. However, with growing database size, the performance advantage of TRENCH over rank-shrink leads to consistently better results.

In general, TRENCH runtimes grow proportional with $|U|$. Now, we study the influence of restriction g.

7.2.2 Landslides with varying *g*

We have conducted a second evaluation to show that the median performance of TRENCH does not only exceed that of rank-shrink independent of $|U|$, but also for different result size restrictions g. To this end, we have run a set of experiments, where both algorithms crawl the LANDSLIDES dataset over the namespace *event_title* for 8 restrictions ranging from $g = 15$ to $g = 50$ with increments of 5. The outcomes are shown in Fig. 2 (EV2).

As in 7.2.1, TRENCH needs about 10% less queries than rank-shrink to crawl U_{hd}. Notably, the fluctuation is low here and TRENCH consistently outperforms rank-shrink for all evaluated values of g.

During the evaluation, we observed that the count of TRENCH queries correlates with the ratio of $|U|/g$. For example, TRENCH required 337 queries to crawl LANDSLIDES with $|U| = 11033$, $g = 85$ and $|U|/g \approx 130$. Crawling NAMES$_{1500}$ with $|U| = 6494$, $g = 50$ and $|U|/g \approx 130$ required 347 queries. We found several such pairs, where similar $|U|/g$ led to similar query counts. Thus, we did a third evaluation to analyze the effect of the $|U|/g$ ratio on the necessary number of queries. The outcomes are shown in Fig. 2 (EV3).

All runs with the same $|U|/g$ ratio led to similar amounts of queries with fluctuations due to pseudo-random behavior of the foreign API. These results indicate a general runtime of $O(|U|/g)$ and confirm existing complexity studies for this problem class [18].

7.3 Evaluating MINCORE

Since the execution of MINCORE is bound to the processing of user requests, we cannot evaluate it with a single execution like TRENCH. Instead, we simulate continuous user requests and changes to U_{hd} to evaluate the performance of resulting data maintenance tasks. Here, we compare the amount of update queries required by MINCORE to those required by full rebuilds of U_{loc} via TRENCH$_{full}$.

7.3.1 Evaluation method

An evaluation method for the case of a changing, hidden database has been proposed by [11]. Accordingly, we aggregate requests and changes into an artificial time unit *Artificial Day (AD)*.

To initialize the dataset, we randomly transfer half of its entries to U_{hd} and the other half to a separate *entry pool* P. P serves to subsequently extend U_{hd} for simulating the creation of new entries in the hidden database. Correspondingly, we simulate the deletion of entries by moving them from U_{hd} into the pool.

To consider differences in data popularity we randomly classify some entries as *interesting*. These entries are searched disproportionately often to simulate increased demand for specific data. We classify 1% of all entries permanently as *always-interesting* and another 1% as *daily-interesting* changing each artificial day.

As regards user requests, we deduce their search terms from random entries of U_{hd} and specify different deduction methods concerning *completeness* ($\mathcal{C}$).

Generally, a search term is deduced from some value x by removing all characters after an index i. For *normal completeness* ($\mathcal{C}_N$), i is a random, natural number generated using a normal distribution and $0 \leq i \leq length(x) - 1$. We generate a normal distribution from uniformly distributed, random numbers by using the Box-Muller transform [6].

In contrast, for *extensive completeness* ($\mathcal{C}_E$), we set $i = \lfloor log_2(n) \rfloor$. The random natural number n is generated by a uniform distribution with $1 \leq n \leq 2^{length(x)}$. For illustration, consider the exemplary value $test$. It has a 50%, 25%, 12.5% and 12.5% probability of being transformed into $test$, tes, te and t respectively.

We generally transform the (partial) search terms into wildcard queries. That is, a filter a generated for search term m matches any entry, whose value on the searched volume dimension starts with m.

For each experiment, U_{loc} initially holds all entries of U_{hd} and a perfect, up-to-date MINCORE exists for each queryable volume dimension. We perform the following actions sequentially to simulate one AD[5]:

1. Randomly chose 1% of entries as daily-interesting.
2. Randomly insert 3% of entries into U_{hd}.
3. Randomly delete 2% of entries from U_{hd} (excluding interesting entries).
4. Send a specific amount of requests, where 20% of search terms stem from interesting entries.

As said before, we evaluate MINCORE against periodic runs of TRENCH$_{full}$. For comparison, we run both algorithms on the same database but do not write the results of TRENCH$_{full}$ to U_{loc}. This way, both operate under the same conditions without interfering. Generally, we configure our system to run (for MINCORE or TRENCH$_{full}$) with $g = 50$ and $p = 10$.

All experiments last eleven AD with a threshold of three AD for data to become outdated. TRENCH$_{full}$ was scheduled every three AD similar to the timespan t for splinters to become outdated. Note, that for all cases data remains up-to-date during the first three AD and no update queries are send. Therefore, we generally skip the first three AD in all presentations of results.

[5] All percentual values relate to the initial size of U_{hd}.

7.3.2 Varying request patterns

The first evaluation covers varying request patterns, call amounts and search term completeness based on the LANDSLIDES dataset. We combine 100 and 10000 requests per AD with $\mathcal{C}_N$ and $\mathcal{C}_E$ resulting in 4 experiments: $\mathcal{C}_N 100$, $\mathcal{C}_N 10000$, $\mathcal{C}_E 100$, $\mathcal{C}_E 10000$. Requests are made on the volume dimension $source\_name$ as prefix search. The results are shown in Fig. 3 (EV4).

As expected, MINCORE requires significantly less queries than TRENCH$_{full}$ to serve the latest entries for all user requests. Hereby, search term completeness turned out to be just a marginal factor of performance. Primarily, the number of update queries can be observed to grow about logarithmically with the amount of user requests.

For explanation first note that a finite number of splinter updates suffices to achieve volume consistency over the whole namespace. Thereafter, user requests can be served without a need for further update queries. When splinters become outdated again, user requests are causing an increase of update queries until all splinters are up-to-date again. For high amounts of user requests (10000/AD) almost all updates are triggered immediately and thus occur on the following day (AD 4, 7, 10). For low amounts of user requests (100/AD), updates grow much slower and occur continuously as not all outdated splinters are ever required.

We note that using the namespace $source\_name$ is unfavorable for showing MINCORE's performance, since the average length of such (string) values is rather low. As a result, the search filters deduced from values of $source\_name$ cover a wide range and therefore intersect a large amount of splinters. We have also performed the evaluation EV4 using the namespace $event\_title$, whose (string) values are much longer. This leads to user requested search queries with rather narrow ranges and therefore less updates are triggered. For example, $\mathcal{C}_N 100$ and $\mathcal{C}_E 100$ lead to 402 and 350 update queries when using $source\_name$, but only 330 and 317 queries when using $event\_title$. However, our focus in this evaluation is on interactive application scenarios where human users are involved. For such cases, we consider the use of very long search terms, like those generated from $event\_title$, as unrealistic. Therefore, all evaluations regarding MINCORE use the $source\_name$ namespace in order to simulate an AD as realistic as possible. More diverse application scenarios are likely to show even better performance.

For a second evaluation, we analyze whether the performance of MINCORE depends on $|U|/g$ as observed for TRENCH. We therefore vary the size of U_{hd} by changing datasets to NAMES$_{9100}$ and NAMES$_{5280}$ with 59920 and 30016 entries while keeping $g = 50$. Additionally, we increase the growth rate of U_{hd} with 10% insertions and 1% deletions per AD. We restrict search term completeness to

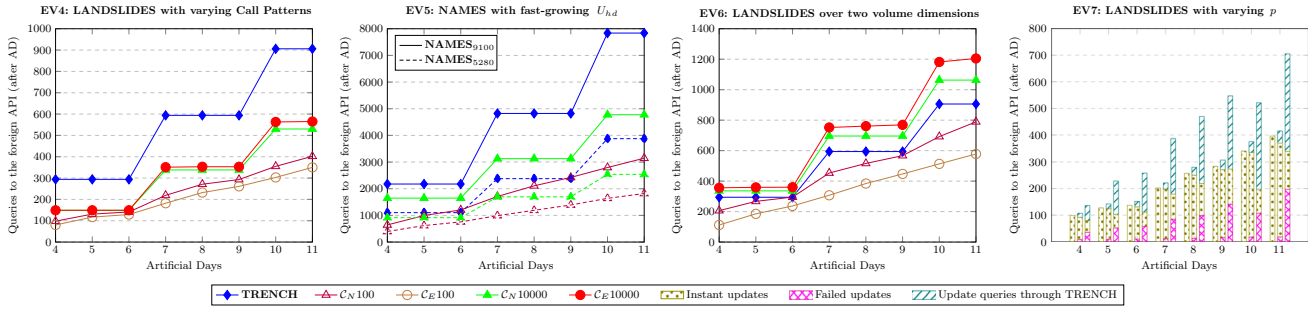

Fig. 3 Evaluation results for MINCORE

$\mathcal{C}_N$, as it turned out to be less significant. The results are shown in Fig. 3 (EV5).

The outcomes show that changing $|U|$ directly impacts the performance of MINCORE, even if all other factors, including g, are equal. This indicates that the performance dependents on the value of $|U|/g$. Furthermore, MINCORE still outperforms TRENCH$_{full}$ even for the worst case. Thus, MINCORE is not only superior to TRENCH$_{full}$ for small and rather static hidden databases, but also for large ones with high change rate.

A third experiment evaluates the performance of MINCORE as regards the dimensionality of data. Remember that a replication service might support user requests over multiple volume dimensions with separate MINCOREs. Each MINCORE increases the number of namespaces that have to be maintained through regular update queries. Moreover, querying an entry d over some volume dimension v influences all MINCORE over any volume dimension containing a value of d.

Therefore, we measure the performance impact of running multiple MINCORE and distributing user requests over their volume dimensions. More precisely, we repeat EV4 but split user requests regarding two volume dimensions $event\_id$ and $source\_name$ with separate MINCORE over each. The results are shown in Fig. 3 (EV6).

Obviously, the amount of update queries increases significantly. In fact, while MINCORE is still better for $\mathcal{C}_{N/E}100$, TRENCH$_{full}$ is more efficient for $\mathcal{C}_{N/E}10000$. This is due to the fact that TRENCH$_{full}$ gathers all entries of the database by completely crawling the universal namespace. MINCORE only queries small ranges of namespaces and cannot reason on the volume consistency of any other volume dimension than its own.

In conclusion, our evaluations of MINCORE show that the approach is not universally superior to regathering a hidden database. Yet, it is favorable when dealing with low-requested hidden databases, where requests spread over few volume dimensions.

7.3.3 Adjusting the buffer size

Another interesting aspect is the impact of buffer size p on MINCORE performance. In general, p adjusts navigators to query less than g-many entries. While this increases the rate of volume consistent query results, it also increases the number of initial update queries. We analyze this effect by comparing the overall number of update queries for p_1=10, p_2=5 and p_3=0. Experiments are based on LANDSLIDES with 10% insertions and 1% deletions per AD, a limit of $g = 50$ and the $\mathcal{C}_N100$ request pattern.

The results are shown in Fig. 3 (EV7). It shows three bars each AD for p_1, p_2, p_3. Each one is further divided as regards queries that have instantly succeeded or failed and thus led to further TRENCH queries.

Clearly, even a small buffer leads to much less volume inconsistent results and causes a significant decrease of queries. About 28% off all navigator queries lead to a volume inconsistent result with $p = 0$, shrinking to approximately 4% for $p = 5$ and zero for $p = 10$.

7.3.4 Measuring impacts on data consistency

Some use-cases might prefer a small error in their responses, if that lowers the amount of queries sent to the foreign API in exchange. Generally, one can lower the overall number of update queries by extending the validity time range of splinters and thus increasing acceptable staleness. Thus we study the impact of splinter validity on a response's data consistency and the amount of update queries sent to the foreign API.

To this end, we use the same setup as EV5 with the dataset NAMES$_{5280}$ and the request-pattern $\mathcal{C}_N100$. We measure the data consistency of a query A by its error $|Q_{loc}(A) \setminus Q(A) \cup Q(A) \setminus Q_{loc}(A)|$, which counts the missed or wrongly returned entries, as well as its correctness $|Q_{loc}(A) \cap Q(A)|$, which counts the correctly returned entries. Hereby, we will identify entries by their value on the volume dimension v_{unique}.

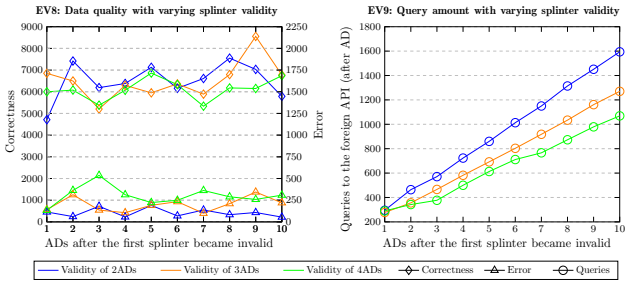

Fig. 4 Impact of varying splinter validity regarding data quality and performance

We have executed the evaluation for splinter validities of two, three and four days. Measurements started at the AD after the first splinter became invalid, which otherwise differs for increasing splinter validities. The measured error and correctness, as well as the number of queries sent to the foreign API, are shown in Fig. 4 (EV8 and EV9).

Increasing the staleness decreases the overall consistency of the returned data. The average error increases from 103 at a validity of two days to 276 at 4 days, as seen in evaluation EV8. On the other hand, the decrease in queries displayed in EV9 shows to be significant. The number of queries sent to the foreign API drops by roughly a third when increasing the validity from two to four days. In the end, it has to be decided on a case-by-case basis, if higher data consistency or less update queries are more preferable for a use case.

7.4 Showing real-world feasibility

So far, experiments have been conducted under lab conditions with locally controlled databases simulating foreign API behavior. This was necessary to analyze effectiveness and efficiency of our algorithms. However, it does not show the feasibility of our approach under real-world conditions. We now present a field study building on an unassociated, real-world hidden database. Thereby, we shift the focus to different measures, namely running times, to show practical utility.

For the field study, we have used a public LDAP API provided by NCSU (see 7.1). In particular, we have selected the LDAP domain for student data[6] that promised to offer a sufficient number of entries. An important goal was to control the resulting load of the public directory service and keep it on a reasonable level. Therefore, we have restricted the extend of replication to a range $[a, h)$ of entry name prefixes. Furthermore, we have not persisted any of the public data in the course of this study.

For implementation, we set up the HDRS-prototype (see 6.2) to query the NCSU LDAP service for entries on the volume dimension *sn* (surnames). Since we expected only marginal changes of U per day, we set the buffer size to a rather small value of $p = 10$. Furthermore, we simulated a $\mathcal{C}_N 100$ user request pattern on *sn* for one real week. The results are shown in Table 1.

[6] Accessible at ldap://ldap.ncsu.edu:389/ou=students,ou=people, dc=ncsu,dc=edu.

Table 1 Results of NCSU field study

d	q	t_s	t_e	t_t	e	s	r_l
0	72	7s	306s	313s	14093	30	-
1	30	20s	139s	159s	14095	31	72
2	29	18s	136s	154s	14112	38	74
3	32	21s	129s	150s	14103	38	69
4	32	19s	130s	149s	13039	36	70
5	29	33s	119s	152s	12985	31	74
6	28	17s	113s	130s	12983	32	74
7	28	19s	108s	127s	12983	31	74

d:day, q:update queries, t_s:time system, t_e:time extern, t_t:time total, e:entries ($|U_{loc}|$), s:splinters, r_l: requests against U_{loc} w/o updates

The initial execution of TRENCH on $[a, h)$ required 72 queries and returned a total of 14093 entries. For that purpose, the algorithm required 313 seconds including 90 seconds of waiting for responses from NCSU and 7 seconds of local computing. To reduce the query load for NCSU, we have included 3 seconds of artificial pause after each query summing up to 216 seconds.

Subsequently, MINCORE required an average of 30 update queries to serve the 100 user requests per day. This amounts to approximately 59% less queries than re-crawling $[a, h)$ with TRENCH would have caused. On average, 100 user requests got answered in about 140 seconds, which corresponds to an average time of 1.4 seconds to answer per request. However, a major amount of this time consisted of waiting for responses from NCSU and artificial pauses. Computing time of our system contributed an average of 21 seconds overall or 210 milliseconds per request.

An interesting observation can be made regarding the update of day four, which was conducted on the 30th of January 2020. Here, over 1000 entries got removed from U_{loc} by MINCORE. It can be speculated that a monthly cleanup procedure removed old entries from NCSU and this caused the significantly lower entry amount after the update. This hypothesis is supported by the fact that day four was the only outlier regarding the daily change of $|U_{loc}|$. All other updates, before and after day four, consistently changed the size of U_{loc} by only a few entries. That is, the entry count did not bounce back to its former average value but stayed at about 13000 entries during days five to seven.

Update queries were caused by 28% of user requests on average. Under the assumption that no ranges were requested while being updated, 72% of requests on average were

answered without waiting for responses from NCSU. Alternatively one could choose not to wait for responses from NCSU at all by answering requests directly from U_{loc} before finishing the updates. Obviously this would decrease the average response time from a few seconds to mere milliseconds at the cost of consistency. In the end, the choice depends on individual application requirements.

8 Conclusion and outlook

Hidden databases can be found throughout the web and beyond. They offer data sources that are often without alternative, e.g., domain-specific or real-time data in the context of the social web, electronic business, public administration, cyber-physical systems and many others. In doing so, hidden databases focus on interactive access patterns and are otherwise heavily restricted.

In this paper we have pursued the goal of making hidden database contents permanently accessible without limitations. To this end, we have proposed an independent replication service. The idea is to replicate a hidden database and retain its interface, but to overrule the ranked retrieval model and query rate restrictions. Thus, issuing unlimited amounts of queries including full scans becomes possible and inexpensive.

Summary We have presented two fundamental algorithms 1) TRENCH to crawl and 2) MINCORE to update a hidden database over its public remote API. The TRENCH algorithm enables to crawl any parts of a hidden database for initiating or updating a local copy despite remote API restrictions. It adopts a flexible crawling strategy to find the most efficient ranges covering an unknown dataset and overcomes very dense areas by lower dimension search (LODIS).

The MINCORE algorithm enables to keep the local copy consistent with the dynamic hidden database in the light of autonomy and access restrictions. Therefore, it adopts a client-centric consistency model that reduces updates to those parts of the local copy that are actually being requested based on staleness. This approach aims to optimize mixed generic access patterns. For the worst case of sole full scans, it falls back to the baseline. Yet, its efficiency increases with more specific user queries and diversity of data popularity.

Beyond the conceptual results, we have presented HDRS, a cloud-based service that coordinates and executes the introduced algorithms in order to provide hidden database replication as a service. We have implemented the proposed architecture as regards prototypes of the algorithms as well as a full-fledged service system on the AWS platform. Based on these systems we conducted a variety of experiments for evaluation. These showed clear improvements of our crawling and update algorithms over current state-of-the-art solutions in many cases of laboratory-based experiments and a real-world field study.

Discussion TRENCH builds on existing work from the field of hidden database crawling. However, its novel approach of including entries from U_{loc} into the calculation of further crawling steps leads to a better performance than rank-shrink, the former state of the art that calculates crawling steps based solely on the last query's result. We believe that dynamically calculating further values, such as $step$, based on U_{loc} during runtime might boost performance even more. In terms of updating hidden database snapshots, MINCORE is the first of its kind. Its performance has been shown to clearly surpass the naive baseline algorithm. Both, TRENCH and MINCORE provide generic contributions that can be adopted in many areas like aggregate functions, data sampling, query re-ranking and others.

To the best of our knowledge a generic replication service for hidden databases has not been proposed before in this form. In particular, HDRS opens up a novel practical application area for interactive use cases. Beyond the prototypical evaluation with a real-world hidden database, we just briefly mention its successful internal application for intranet use cases. The generic HDRS prototype is available as open source, because we believe that it can provide similar benefits in many more cases.

Future Work So far, we evaluated our replication approach for interactive application scenarios, where a hidden database copy is used to answer search queries of human users. In the future, we would like to include mixed application scenarios that include different request patterns, e.g., for data analytics. In terms of fundamental features, a possible direction would be to extend MINCORE as well as LODIS over multiple volume dimensions. Also the generic approach might be augmented with respect to various specific situations like handling massive data changes similar to [20].

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Álvarez M, Raposo J, Pan A, Cacheda F, Bellas F, Carneiro V (2007) DeepBot: A focused crawler for accessing hidden web content. In: ACM International Conference Proceeding Series, vol. 236, pp. 18–25. ACM
2. Alwan AA, Ibrahim H, Udzir NI, Sidi F (2013) Estimating missing values of skylines in incomplete database. In: Proceedings of the 2th International Conference on Digital Enterprise and Information Systems, pp. 220–229. SDIWC
3. Asudeh A, Zhang N, Das G (2016) Query reranking as a service. Proceedings of the VLDB Endowment 9(11):888–899
4. Barbosa L, Freire J (2007) An adaptive crawler for locating hiddenwebentry points. In: C.L. Williamson, M.E. Zurko, P.F. Patel-Schneider, P.J. Shenoy (eds.) Proceedings of the 16th international conference on World Wide Web – WWW '07, p. 441. ACM Press
5. Barbosa L, Freire J (2010) Siphoning hidden-web data through keyword-based interfaces. Journal of Information and Data Management 1(1):133–144
6. Box GEP, Muller ME (1958) A Note on the generation of random normal deviates. The Annals of Mathematical Statistics 29(2):610–611
7. Durairaj Gunasekaran Y, Asudeh A, Hasani S, Zhang N, Jaoua A, Das G (2018) QR2: A Third-Party Query Reranking Service over Web Databases. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 1653–1656. IEEE
8. Jin X, Zhang N, Das G (2011) Attribute domain discovery for hidden web databases. In: Proceedings of the 2011 international conference on Management of data – SIGMOD '11, p. 553. ACM Press, New York, New York, USA
9. Kambhampati S, Wolf G, Chen Y, Khatri H, Chokshi B, Fan J, Nambiar U (2007) QUIC: Handling query imprecision & data incompleteness in autonomous databases. In: CIDR 2007 – 3rd Biennial Conference on Innovative Data Systems Research, pp. 263–268
10. Kumar M, Bindal A, Gautam R, Bhatia R (2018) Keyword query based focused web crawler. Procedia Computer Science 125:584–590
11. Liu W, Thirumuruganathan S, Zhang N, Das G (2014) Aggregate estimation over dynamic hidden web databases. Proceedings of the VLDB Endowment 7(12):1107–1118
12. Lu Y, Thirumuruganathan S, Zhang N, Das G (2015) Hidden database research and analytics (hydra) system. IEEE Data Eng. Bull. 38(3):84–102
13. Madhavan J, Ko D, Kot Ł, Ganapathy V, Rasmussen A, Halevy A (2008) Google's Deep web crawl. Proceedings of the VLDB Endowment 1(2):1241–1252
14. Meng X, Ma ZM, Yan L (2009) Answering approximate queries over autonomous web databases. In: Proceedings of the 18th international conference on World wide web – WWW '09, p. 1021. ACM Press, New York, New York, USA
15. Nguyen H, Nguyen T, Freire J (2008) Learning to extract form labels. Proceedings of the VLDB Endowment 1(1):684–694
16. Rezk E, Aqle A, Jaoua A, Das G, Zhang N (2017) Optimized Processing of a Batch of Aggregate Queries over Hidden Databases. In: 2017 International Conference on Computer and Applications (ICCA), pp. 317–324. IEEE, IEEE
17. Savković O, Mirza P, Tomasi A, Nutt W (2013) Complete approximations of incomplete queries. Proceedings of the VLDB Endowment 6(12):1378–1381
18. Sheng C, Zhang N, Tao Y, Jin X (2012) Optimal algorithms for crawling a hidden database in the web. Proceedings of the VLDB Endowment 5(11):1112–1123
19. Song S, Zhang A, Chen L, Wang J (2015) Enriching data imputation with extensive similarity neighbors. Proceedings of the VLDB Endowment 8(11):1286–1297
20. Suhaim SB, Liu W, Zhang N (2016) Discover Aggregates Exceptions over Hidden Web Databases. arXiv preprint arXiv:1611.06417
21. Wolf G, Kalavagattu A, Khatri H, Balakrishnan R, Chokshi B, Fan J, Chen Y, Kambhampati S (2009) Query processing over incomplete autonomous databases: Query rewriting using learned data dependencies. The VLDB Journal 18(5):1167–1190
22. Yu H, Vahdat A (2002) Design and evaluation of a conit-based continuous consistency model for replicated services. ACM Trans. Comput. Syst. 20(3):239–282
23. Zhao F, Zhou J, Nie C, Huang H, Jin H (2016) SmartCrawler: A two-stage crawler for efficiently harvesting deep-web interfaces. IEEE Transactions on Services Computing 9(4):608–620