VFDS: An Application to Generate Fast Sample Databases

Conference Paper · November 2014

DOI: 10.1145/2661829.2661845

4 authors, including:

Teodora Sandra Buda
IBM
22 PUBLICATIONS 132 CITATIONS
SEE PROFILE

Thomas Cerqueus
Lengow
37 PUBLICATIONS 175 CITATIONS
SEE PROFILE

John Murphy
University College Dublin
234 PUBLICATIONS 2,659 CITATIONS
SEE PROFILE

VFDS: Very Fast Database Sampling System

Teodora Sandra Buda
Lero, Performance Engineering Laboratory
Computer Science and Informatics
University College Dublin
teodora.buda@ucdconnect.ie

Thomas Cerqueus
Lero, Performance Engineering Laboratory
Computer Science and Informatics
University College Dublin
thomas.cerqueus@ucd.ie

Morten Kristiansen
IBM Collaboration Solutions
IBM Software Group
Dublin, Ireland
Morten_Kristiansen@ie.ibm.com

John Murphy
Lero, Performance Engineering Laboratory
Computer Science and Informatics
University College Dublin
J.Murphy@ucd.ie

***Abstract*—In a wide range of application areas (e.g. data mining, approximate query evaluation, histogram construction), database sampling has proved to be a powerful technique. It is generally used when the computational cost of processing large amounts of information is extremely high, and a faster response with a lower level of accuracy for the results is preferred. Previous sampling techniques achieve this balance, however, an evaluation of the cost of the database sampling process should be considered. We argue that the performance of current relational database sampling techniques that maintain the data integrity of the sample database is low and a faster strategy needs to be devised. In this paper we propose a very fast sampling method that maintains the referential integrity of the sample database intact. The sampling method targets the production environment of a system under development, that generally consists of large amounts of data computationally costly to analyze. We evaluate our method in comparison with previous database sampling approaches and show that our method produces a sample database at least 300 times faster and with a maximum trade off of 0.5% in terms of sample size error.**

I. Introduction

With every new release of a product, the implementation of a change request, or a change in the configuration of a deployment, a series of similar tests need to be carried out in order to make sure that the core functionality of the system remains intact [1]. Moreover, 60% of total software development costs are devoted to enhancing existing applications, to add or modify functionality, rather than developing new applications [2]. This means that it is reasonable to expect that in many projects an operational database exists from which the sample data can be extracted (e.g. Web-enabling existing applications). However, generally these databases consist of large amounts of data, which are costly to analyze. As databases increase over time, certain concerns have to be considered, such as scalability, storage space, network and power consumption. Database sampling from the operational data available is a potential solution to overcome these challenges and provide a realistic testing environment.

Database sampling has a long history in computer science, proving its usefulness in numerous scenarios where using the entire database is infeasible because of the complexity of handling large amounts of data. In these situations, a compromise has to be achieved in order to analyze the dataset faster and generally a subset of the data is preferred. Some benefits of sampling large databases are: significantly decrease the storage space for the testing environment, decrease the administration overhead of managing datasets for the testing environment, and nevertheless increase the computational efficiency of running the tests using a smaller database. Moreover, sampling from the production environment will determine the sample contains realistic test data, encompassing a variety of scenarios the user created, and serving as an invaluable input for testing the core functionality of the system under development. However, current practices of relational database sampling while preserving the integrity of the data in the sample database are computationally costly [3], [4]. This cost is the first objective that database sampling is trying to avoid: a faster approach should be devised.

Database sampling aims to: (i) reduce the size of the original database, (ii) decrease the cost of maintenance for the target database, (iii) decrease the cost of the following analysis (e.g. software validation, approximate query answering) to be applied on the sample. In this paper we propose a very fast database sampling system called VFDS, with the objective of solving all the above mentioned challenges. The main contribution of the proposed sampling method is the high speed at which the sample database is produced. The high performance of the method makes VFDS a suitable candidate for real-time systems that perform various analysis on a database. For instance, VFDS can be used as a sampling method for providing fast approximate answers for users' queries on a real-time application. Furthermore, it can significantly reduce the computational cost of running data mining algorithms on the sample dataset, without the need of an initial time-consuming setup. Moreover, it can drastically reduce the amount of time needed for generating test database by using a sample database of the production environment. Results show that VFDS performs at least 300 times faster than previous approaches, with a trade-off of maximum 0.5% between VFDS and the best method in terms of sample size error. In addition, results show that VFDS provides similar approximate query answers with previous approaches.

This paper is organized as follows: Section II discusses related work in database sampling and data population of testing environments. Section III describes the proposed VFDS system. In section IV, we evaluate our method in comparison with previous approaches. Section V concludes the paper.

II. Related Work

Existing methods for constructing test databases generally focus on generating data by combining the database schema definition with conflict resolution, disclosure assessment and data perturbation methods (e.g. additive noise approach) [5] and randomized functions for populating testing environments [6], [7]. In [8], the authors focus on techniques to optimize the generation of test databases by using sequential and parallel algorithms given the statistical distribution of the underlying data. Other generic tools generate synthetic data using the schema only [9], [10]. In [1] the reader is presented with a different approach than the ones discussed before for generating test databases. The tool uses RQP (Reverse Query Processing). It receives as input the test queries and the schema of the target database and generates as output the data for the testing environment. However, synthetic data is dependent on the parameters given, and using it for populating the testing environment can result in missing important test cases for the system under development. In [11] the authors try to adjust the synthetic data generation focusing on the relevance of the production database for testing purposes. The method proposed initially studies the properties of the production database and generates data according to patterns discovered in the production database. In [12] the reader is presented an anonymization framework based on constraint satisfaction, emphasizing on the advantages of using production data in the testing environment. Unfortunately, the production environment generally consists of large amounts of data that are computationally costly to analyze.

From the existing sampling approaches, Olken's major contribution to random sampling from large databases proves sampling to be a powerful technique [13]. A random sampling technique involves randomly selecting tuples from each table from the original database (i.e. not following any selection criteria). However, this subset of the original dataset should respect the rules defined in the original database in order to maintain the integrity and quality of the data stored (e.g. foreign key constraints in a relational database). Sampling from single-table databases is a popular technique used in the data mining community [14], generally applied on the UCI Machine learning repository[1]. However, most of today's structured data is stored in relational databases, where data is stored in multiple tables connected through various constraints. Bisbal's approach extends table-level sampling and proposes a consistent database sampling technique targeting relational databases, and focusing on the advantage of using prototype databases populated with sampled operational data [15]. Data is selected following various constraints (e.g. domain constraint, referential constraint) such that the sample is consistent with the original database. Another sampling approach that extended the table-level sampling to relational database sampling is presented in [4]. The authors propose a sampling mechanism called *Linked Bernoulli Synopses* based on *Join Synopses* [3] aiming to provide fast approximate query answers for queries that contain joins over multiple tables. Their solution involves maintaining the foreign key integrity of the synopses and is oriented towards approximate query answering. A more recent work in the database sampling community is described in [16], where the reader is presented a representative sampling approach aimed to handle scalability issues of processing large graphs. However the approach is oriented towards graph-structured data.

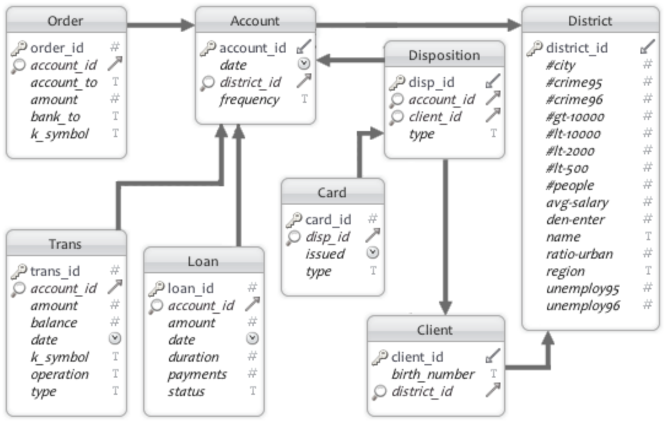


Fig. 1. The *Financial* relational database.

III. VFDS

In this section, we describe the VFDS (Very Fast Database Sampling) system which is designed to produce a database sample from a relational database. The objective is to maintain the data integrity of the database, while speeding up the database sampling process. In this work, we make the following assumptions: (i) the database is normalized in the third normal form, and (ii) there are no cycles of dependency between the tables.

The system produces the sample database in only one run over the entire database and consists of two phases. VFDS receives as an input from the user the sampling rate to apply on the original database. In the first phase, the system selects a starting table according to the impact of the table on the sampling process (section III-B1). The system proceeds in randomly sampling tuples from the selected table according to the sampling rate (section III-B4). The second phase of the system consists of sampling the tuples associated (i.e. referencing and referenced) with the already inserted tuples in the sample database (section III-B5).

[1]http://archive.ics.uci.edu/ml/

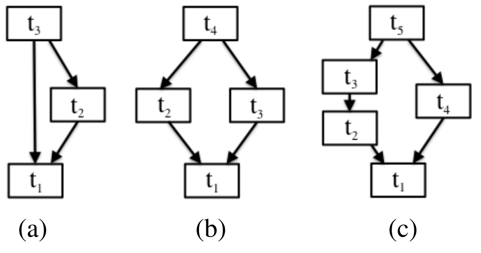


Fig. 2. Examples of different diamond patterns.

We present the complete approach in section III-B, after describing in detail the notations we used for this approach.

A. Notations

We denote the original database by $O(T)$, where T represents the set of tables from the original database. Fig. 1 presents an example of a relational database with 8 tables. A sample database, denoted by $S(T)$, respects the same schema as $O(T)$ and contains a subset of the data contained in the original database.

We refer to the tuples of table t_i in the original and the sample database with $O(t_i)$ and $S(t_i)$. A table t_i consists of various attributes. A non-empty set of them form the set of primary keys, denoted by $PK(t) = \{pk_1, pk_2, ..., pk_m\}$, and a set of attributes involved in foreign keys constraints, $FK(t) = \{fk_1, fk_2, ..., fk_q\}$. We denote by $FK_j(t_i)$ the set of attributes of table t_i that reference table t_j. In other words, a table t_i references table t_j if exists $fk_i \in FK_j(t_i)$ such that fk_i points to $pk_j \in PK(t_j)$. We denote this relationship by $t_i \rightarrow t_j$. Symmetrically, t_j is referenced by t_i and this is denoted by $t_j \leftarrow t_i$. We refer to the children of table t by $children(t)$, representing the set of tables that t references: $children(t) = \{t_i \in T : t \rightarrow t_i\}$. Moreover, $desc(t)$ refers to the set of all the descendants of t. It is defined recursively by:

$$desc(t) = children(t) \cup \bigcup_{t_i \in children(t)} desc(t_i)$$

Similarly, we refer to the parents of t by $parents(t)$, representing the set of tables that reference table t: $parents(t) = \{t_i \in T : t \leftarrow t_i\}$. We refer by $ances(t)$ to the set of all the ancestors of t:

$$ances(t) = parents(t) \cup \bigcup_{t_i \in parents(t)} ances(t_i)$$

We compute the distance between two tables t_i and t_j as the number of foreign keys that need to be traversed to reach t_j from t_i. We denote by $commonAnc(t_i, t_j)$ the closest common ancestor between t_i and t_j. For instance, in Fig. 1, $commonAnc(Account, Client)$ = $Disposition$.

A *diamond pattern* appears when two tables, t_i and t_j such that $t_i \notin desc(t_j) \cup ances(t_j)$ and $t_j \notin desc(t_i) \cup ances(t_i)$, have a common ancestor and a common descendant. We denote a diamond pattern as a sequence of table $\langle t_1, ..., t_s \rangle$ where $s \geq 4$. Table t_1 is called the *base*, t_s is called the *summit*, and $t_2, ..., t_{s-1}$ are called *intermediate tables* of the diamond. A diamond pattern can also appear between three tables t_1, t_2 and t_3, when $t_3 \rightarrow t_2$, $t_2 \rightarrow t_1$ and $t_3 \rightarrow t_1$. Fig. 2-a presents such a diamond. Examples of other diamonds are depicted in Fig.2-b and 2-c. In Fig. 1, we can identify the diamond pattern $\langle District, Account, Client, Disposition \rangle$. *District* is the base, *Disposition* is the summit, and *Account* and *Client* are intermediate tables.

Finally, we denote by α the *sampling rate* received from the user with a value in $[0, 1]$.

B. Approach

1) Starting table selection: The starting table, denoted by $t_\star$, critically impacts the resulting sample database as the method only samples its directly and indirectly associated tuples. By selecting a table with high number of associated tuples, the method increases the probability of meeting the space constraints sent by the user. Moreover, the number of tuples of the $t_\star$ contributes to the impact of the starting table. Thus, the system selects as $t_\star$ the table with the maximum number of related tuples (i.e. number of tuples of the starting table together with the number of distinct tuples from the associated tables). For this reason, in comparison with previous approaches that employ a top-down approach (e.g. [3], [4]), VFDS employs both a top-down and bottom-up approach depending on the starting table. Algorithm 1 presents the starting table selection.

Algorithm 1: startingTableSel(T)

```
1 $max \leftarrow 0$;
2 for $t \in T$ do
3     $count \leftarrow$ countTuples($t$) ;
4     for $t_i \in children(t) \cup parents(t)$ do
5         $count \leftarrow count +$ countRefTuples($t, t_i$);
6     if $max < count$ then
7         $max \leftarrow count$;
8         $t_\star \leftarrow t$;
9 return $t_\star$;
```

The function `countTuples(t)` (algorithm 1, line 3) counts all the tuples of table t. The function `countRefTuples(t,t_i)` (algorithm 1, lines 4 and 5) counts all the distinct tuples of table t_i that reference or are referenced by table t. The function executes the following MySQL query:

```
SELECT COUNT(DISTINCT $t_i.PK_i$) FROM $t \bowtie t_i$
```

2) Graph construction and diamond discovery: In this pre-processing phase, the method builds a graph representation of the database under analysis in order to detect the existing diamond patterns in the database. A bottom-up approach is employed from each leaf of the database in order to determine the existing summits, bases, and intermediate tables in the database. A diamond pattern is detected when the following condition is met: $\exists\ t, t_i, t_j, t_s \in T : t_i \notin desc(t_j) \cup ances(t_j) \wedge t_j \notin desc(t_i) \cup ances(t_i) \wedge t_i, t_j \in ances(t) \wedge t_s = commonAnc(t_i, t_j)$. For each table of the database, the method records whether the table is a summit, a base, or an intermediate table.

3) Tuples insertion: The tuples insertion in the sample database is done in two phases: (i) the starting table $t_\star$ is filled, (ii) the other tables are filled with associated tuples of $S(t_\star)$. Algorithm 2 presents these two phases: starting table filling (algorithm 2, lines 1 and 2), and other tables filling (algorithm 2, lines 3 to 4). The latter is performed by inserting tuples in the starting table's children and parents. As `fillTable` is a recursive function, its execution with the starting table's children and parents as parameters triggers the insertion in the other tables of the database (i.e. starting table's descendants and ancestors). The next two sections present in details these two phases.

Algorithm 2: fillSampleDb($t_\star$, α)

```
  // Phase 1:
1 $N \leftarrow \alpha \cdot$ execute("SELECT COUNT(*) FROM $O.t_\star$");
2 fillStartingTable($t_\star$, $N$);
  // Phase 2:
3 for $t_i \in children(t_\star)$ do fillTable($t_i$,$t_\star$);
4 for $t_i \in parents(t_\star)$ do fillTable($t_i$,$t_\star$);
```

4) Insertion in the starting table: The input for this phase is the starting table and a number N of tuples to sample. The output represents a set of tuples from the starting table inserted in the sample database. Tuples from the starting table $t_\star$ are randomly selected for insertion from the original database $O(t_\star)$. Function `fillStartingTable` (algorithm 2, line 2) consists in executing the following query:

```
INSERT INTO $S.t_\star$
SELECT * FROM $O.t_\star$ ORDER BY RAND()
LIMIT $N$
```

In this process, all the tuples of the table have a probability equal to α to be inserted. The previous query ensures that the following condition is met: $\|S(t_\star)\| = \alpha \cdot \|O(t_\star)\|$ where $\|O(t_\star)\|$ and $\|S(t_\star)\|$ represent the number of tuples of the starting table in the original database and in the sample database. The random selection can also be performed by generating N different random numbers with a maximum value of the total number of tuples of $t_\star$ and inserting the corresponding tuples.

5) Insertion in other tables: The objective of this phase is to populate the entire sample database from the tuples inserted in the starting table. The tables are traversed in a depth-first manner and they are filled according to the tuples already inserted in their parents or children. The method includes only the tuples necessary to satisfy foreign key constraints. Through this strategy, the method ensures that the referential integrity of the sample remains intact. The main advantage of this approach is that it minimizes the cost of the process because it is not necessary to process separately all the tuples of the original database.

Function `fillTable(`t_i,t_j`)` (algorithm 2, lines 3 and 4) triggers the insertion in the other tables of the database and is presented in detail in algorithm 3. The function aims at filling table t_i according to the already sampled tuples of table t_j. Depending on the relationship between table t_i and t_j, a different method is called for the insertion. Given two tables t_i and t_j such that $t_i \rightarrow t_j$, the function `fillFromChild(`t_i,t_j`)` (algorithm 3, line 17) consists of filling table t_i according to the set of tuples already inserted in its child table t_j. The following query is used to perform the insertion:

```
INSERT INTO $S.t_i$ (SELECT * FROM $O.t_i$
WHERE $FK_j(t_i)$ IN (SELECT $PK(t_j)$ FROM
$S.t_j$))
```

The query inserts all the tuples of $O(t_i)$ in $S(t_i)$ that reference tuples of $S(t_j)$. Symmetrically, given two tables t_i and t_j such that $t_i \leftarrow t_j$, `fillFromParent(`t_i, t_j`)` (algorithm 3, line 16) builds and executes the following query:

```
INSERT INTO $S.t_i$ (SELECT * FROM $O.t_i$
WHERE $PK(t_i)$ IN (SELECT $FK_i(t_j)$ FROM
$S.t_j$))
```

Let us consider the database presented in Fig. 1. If table *Account* is already filled, the execution of `fillTable(`*Loan*, *Account*`)` consists in populating table *Loan* from the existing tuples in *S(Account)* and in executing the following query:

```
INSERT INTO $S.Loan$ (SELECT * FROM $O.Loan$
WHERE $Loan.account\_id$ IN
(SELECT $Account.account\_id$ FROM $S.Account$))
```

The previous queries are used in the general case but are not suitable when diamond patterns appear because the insertions in a table of a diamond pattern may lead to the insertion of additional tuples in the other tables of the diamond. For instance, let us consider that tables *District* and *Account* are already filled. The insertion of tuples in table *Disposition* will trigger insertion of tuples in *Client*. It will then trigger insertions in *District* (to satisfy foreign key constraints). As a consequence it obliges *District* to store more data. This could lead to an important space overhead. In the case of a diamond pattern, the proposed strategy is slightly different.

6) Diamond pattern insertion strategy: In order to avoid a cyclic flow of insertion when a diamond pattern $\langle t_1, ..., t_s\rangle$ is encountered, the following breadth-first approach is proposed:

- Before inserting tuples in the summit of the diamond t_s, either all children of t_s that belong to the diamond must be previously filled (algorithm 3, line 4), or no child that belongs to the diamond should be filled (algorithm 3, line 7). The latter situation occurs either when the method reaches table t_s first, or when the tables of the diamond can only be reached by firstly populating t_s.
- Before inserting tuples in the base of the diamond t_1, either all parents of t_1 that belong to the diamond must be previously filled (algorithm 3, line 10), or only one parent of t_1 that belongs to the diamond should be filled, if t_s has not been populated yet (algorithm 3, line 13). The latter situation occurs when the tables of the diamond can only be reached by firstly populating an intermediate table t_i, $\forall i \in [2, s-1]$.

The function `fillSummit(`t_i`,` $children(t_i)$`)` is called when inserting tuples in the summit of a diamond pattern (algorithm 3, line 5). Considering that $children(t_i) = \{t_1, \ldots, t_k\}$, the function constructs and executes the following query:

```
INSERT INTO S.t_i
SELECT * FROM O.t_i WHERE
FK_1(t_i) IN (SELECT PK(t_1) FROM S.t_1) AND
...
AND FK_k(t_i) IN (SELECT PK(t_k) FROM S.t_k)
```

For instance, for the diamond pattern {*District, Account, Client, Disposition*} the following query is executed:

```
INSERT INTO S.Disposition
SELECT * FROM O.Disposition
WHERE Disposition.account_id IN
(SELECT Account.account_id FROM S.Account)
AND Disposition.client_id IN
(SELECT Client.client_id FROM S.Client)
```

The function `fillBase(`t_i`,` $parents(t_i)$`)` is called when inserting tuples in the base of a diamond pattern (algorithm 3, line 11). Considering that $parents(t_i) = \{t_1, \ldots, t_k\}$, the function constructs and executes the following query:

```
INSERT INTO S.t_i
SELECT * FROM O.t_i WHERE
PK(t_i) IN (SELECT FK_i(t_1) FROM S.t_1) AND
...
AND PK(t_i) IN (SELECT FK_i(t_k) FROM S.t_k)
```

7) Example of execution: Let us consider the execution of VFDS on the *Financial* database with $t_\star$ = *Account*. VFDS will fill the tables of the database in the following order: *Account*, *District*, *Client*, *Disposition*, *Card*, *Loan*, *Order*, and *Trans*.

Algorithm 3: fillTable(t,t')

```
1  $fillNormal \leftarrow \mathbb{T}$ ;
2  if NOT $isFilled(t)$ then
3      if $isSummit(t)$ then
4          if $\forall t_i \in children(t) : (isInter(t_i)$
           OR $isBase(t_i)) \Rightarrow isFilled(t_i)$ then
5              fillSummit($t$, $children(t)$);
6              $fillNormal \leftarrow \mathbb{F}$ ;
7          else if $\exists t_i \in children(t) : (isInter(t_i)$
           OR $isBase(t_i)) \Rightarrow isFilled(t_i)$ then
8              $fillNormal \leftarrow \mathbb{F}$ ;
9      else if $isBase(t)$ then
10         if $\forall t_i \in parents(t) : (isInter(t_i)$
           OR $isSummit(t_i)) \Rightarrow isFilled(t_i)$ then
11             fillBase($t$, $parents(t)$);
12             $fillNormal \leftarrow \mathbb{F}$ ;
13         else if $\exists t_i \in parents(t) : (isInter(t_i)$
           OR $isSummit(t_i)) \Rightarrow$ NOT $isFilled(t_i)$
           AND $isFilled(commonAnc(t_i, t'))$ then
14             $fillNormal \leftarrow \mathbb{F}$ ;
15     if $fillNormal$ then
16         if $t' \rightarrow t$ then fillFromParent($t$,$t'$);
17         else if $t \rightarrow t'$ then fillFromChild($t$,$t'$);
18     if $isFilled(t)$ then
19         for $t_i \in children(t)$ do fillTable($t_i$,$t$);
20         for $t_i \in parents(t)$ do fillTable($t_i$,$t$);
```

IV. Evaluation

We evaluate our method in comparison with *Join Synopses* (JS) and *Linked Bernoulli Synopses* (LBS) [3], [4]. These two approaches represent to the best of our knowledge the state of the art sampling techniques targeting relational databases that preserve the referential integrity of the sample. Both methods aim at providing fast approximate query answers by producing a sample database of the original large database and applying the queries on the sample instead. As we compare VFDS with JS and LBS, we also consider for the evaluation of VFDS an approximate query answering perspective.

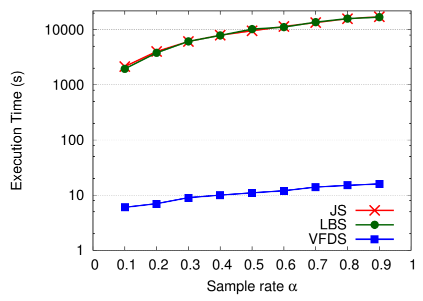
A. JS and LBS overview

Both approaches are probabilistic and require the processing of each tuple in a database. The methods involve retrieving a tuple from the database, assigning a probability of insertion to the respective tuple, and inserting it in the sample database according to this probability. Thus, the methods decide and perform the insertion of each tuple individually. However, the performance of this type of sampling is very poor.

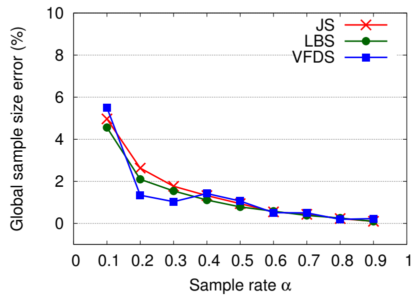
In the case of JS, each tuple from the set of tables will be sampled with a probability equal to α (i.e. the sampling rate) [3]. After this insertion of tuples in the sample database, JS ensures the referential integrity of the sample database remains intact by visiting each of its tables, starting with each identified root, and adding the referenced tuples in the sample database in order to avoid a violation of a foreign key constraint. The decision of whether or not to include the row in the sample is different in LBS [4]. LBS is run only one time over the entire database. LBS requires the retrieval of every tuple from each table and calculates the probability of a tuple, t, being inserted in the sample database based on the probabilities of the tuples referencing tuple t to be inserted in the sample. The computation of this probability is described in detail in [4]. In the case that one of referencing tuples has already been included in the sample, the tuple under analysis is included in the sample with probability 1, thus avoiding the referential integrity to be broken. In the case of a diamond pattern where the table with multiple parents is very small, LBS proposes to store the table completely. Another possibility mentioned by the authors is to switch to JS method.

B. Environment, methodology and database

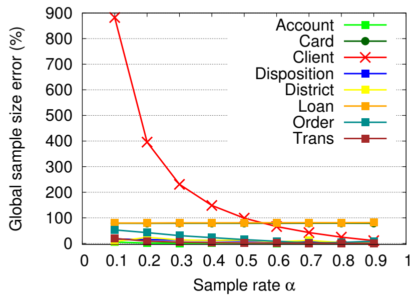
The experiments were run on a machine with quad-core 2.5GHz processor, 16GB RAM, and 750GB Serial ATA Drive with 7200 rpm. VFDS, JS, and LBS were developed using Java 1.6, applied against MySQL databases. Each experiment was run with 12GB maximum size of the memory allocation



(a) Execution time for JS, LBS, and VFDS.



(b) Global sample size error for JS, LBS, and VFDS.



(c) Global sample size error for VFDS depending on the starting table.

Fig. 3. Execution time and global sample size error results.

pool. In order to present representative results, we run each experiment 5 times. The results presented in section IV-D correspond to the average of the different executions.

For this paper, we consider the *Financial*[2] database from PKDD'99 Challenge Discovery, with the schema presented in Fig. 1. The database contains typical bank data, such as its clients information, their accounts, transactions, loans, and credit cards. The database contains 8 tables, and a total of 1,079,680 tuples. The sizes of the tables range from 77 (i.e. in *District* table) to 1,056,320 tuples (i.e. in *Trans* table). The average number of tuples in the database is 134,960. The database occupies 129MB.

C. Metrics

1) Execution time: We measure the execution time needed to sample a database in seconds. The execution time includes the pre-processing phases (computation of probabilities, discovery of the graph of tables, etc.).

2) Sample size error: An accurate sampling method should produce a sample that respects the user's preference in terms of memory constraints. When sampling a database with a sampling rate α, we expect that the original database will be reduced in size by α. As a consequence, we expect the size of the sample database to be equal with the size of the original database reduced by α (i.e. $S_T = \alpha \cdot O_T$). We measure the *global size error* of a sample with respect to a database as:

$$global\_sample\_size\_error(T) = \frac{S_T - \alpha \cdot O_T}{\alpha \cdot O_T}$$

where $O_T = \sum_{t \in T} \|O(t)\|$ and $S_T = \sum_{t \in T} \|S(t)\|$.

3) Query relative error: We use this metric as JS and LBS are systems built for approximate query answering. We measure the accuracy provided by the sample database in answering a query, q, with:

$$query\_relative\_error(q) = \frac{S(q) - O(q)}{O(q)}$$

where $S(q)$, and $O(q)$, represent the result of the query q from the sample database, and from the original database respectively.

[2]http://lisp.vse.cz/pkdd99/Challenge/berka.htm

D. Results

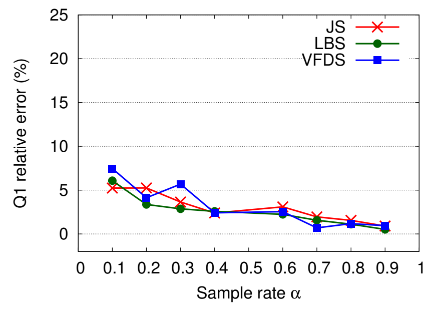
In this section we present the results of running VFDS, JS, and LBS with regards to the metrics described in the previous section. VFDS selects table *Account* as the starting table due to the highest number of related tuples (i.e. 1,073,419). The *Financial* database contains the following diamond pattern: $\langle District, Account, Client, Disposition \rangle$. As table *District* (i.e. table with multiple parents) contains very few number of tuples in comparison with the rest of the tables of the database, we chose to store it completely, as suggested in this case in [4] when applying LBS.

1) Execution time: The execution time of the different methods are presented in Fig. 3-a. We observe that VFDS performs in 11 seconds on average, and JS and LBS perform in 9,761 and 9,742 seconds on average. In the best case scenario for previous methods, i.e. when α is lower than 0.4, we observe the minimum difference between the execution time of VFDS and JS and LBS. This is due to the fact that the number of tuples to insert in the sample database is low and less additional tuples need to be inserted in the sample in order to maintain the referential integrity of the sample. The execution time when $\alpha = 0.1$ is reduced from 2,145 seconds in the case of JS, and 1,949 in the case of LBS to 6 seconds with VFDS. For all the methods we observe that the execution time increases when α increases because of the growing amount of data handled by the methods. We observe that the execution time of JS and LBS increases drastically with the increase of α due to the number of rows that need to be individually assessed for insertion. Thus, the worst scenario for previous methods occurs when $\alpha = 0.9$. In this case, VFDS produces a sample database in 16 seconds, whereas JS produces a sample database in 17,208 seconds, and LBS in 16,883 seconds. As a conclusion, we can say that VFDS drastically reduces the time needed to produce a sample database compared to JS and LBS. In the best case scenario for previous approaches, VFDS performs 300 times faster, while in the worst case it performs 1,000 times faster.

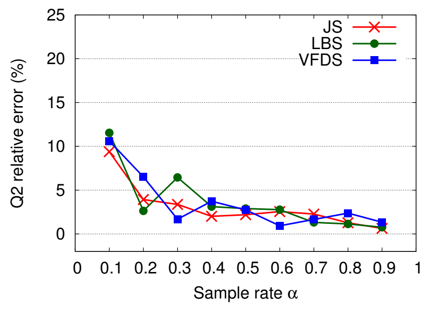
2) Sample size error: The global sample size error can be negative in the case that not enough tuples have been inserted in the sample database. In order to avoid that the positive and negative values of the global sample size error compensate

TABLE I
QUERIES USED FOR ACCURACY EXPERIMENTS.

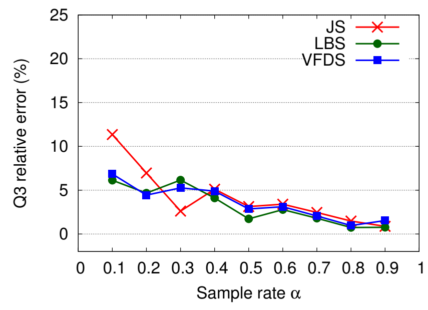
Q1	`SELECT AVG(amount) FROM Loan;`
Q2	`SELECT AVG(payments) FROM Account⋈Loan WHERE Loan.status='A';`
Q3	`SELECT AVG(amount) FROM Account⋈`Order`⋈ District WHERE District.region='west Bohemia';`
Q4	`SELECT AVG(balance) FROM Account⋈Trans⋈Disposition⋈Card WHERE Card.type='classic';`



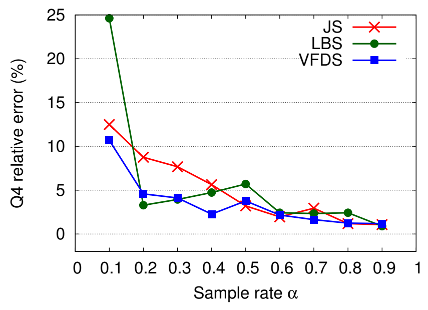
(a) Q1 relative error.



(b) Q2 relative error.



(c) Q3 relative error.



(d) Q4 relative error.

Fig. 4. Query relative error for JS, LBS, and VFDS.

each other over multiple runs, Fig. 3-b presents the average of the absolute value for the global sample size for JS, LBS, and VFDS over the 5 runs. The most accurate method is LBS: the sample size error varies between 4.55% and 0.1% for α between 0.1 and 0.9. We observe that JS samples more data than LBS as the probability of sampling a tuple is higher. The sample size error varies between 4.96% and 0.1% for JS. VFDS is very close to both JS and LBS with the error varying between 5.5% and 0.2%. VFDS provides the best results for $\alpha = \{0.2, 0.3, 0.6, 0.8\}$. The worst case occurs for $\alpha = 0.1$ for all methods due to the fact that the sample size error is relative to the expected sample size. Thus, for small values of alpha, a small variation between the sample and its expected size determines higher values for the error. We notice that the methods provide similar results in terms of sample size error.

Fig. 3-c presents the global sample size error for VFDS depending on the starting table selected by the system. Experimental results show that table *Account* is indeed the starting table that provides the best results in terms of global sample database size. It varies between 5.5% and 0.22%. We observe that table *Client* is the worst candidate as the starting table, with the sample size error varying between 882.4% and 11%. The results are poor when using $t_\star$ =*Client* because for every value of α, the insertion in table *Client* first will trigger the insertion of most rows from the smallest table *District* and then in turn most rows of table *Account*. The latter table will trigger most rows from the largest table *Trans* to be sampled leading to high space overhead in the resulting sample database. Table *Client* has 10,815 related tuples, however, due to the condition described above is not a suitable candidate for the starting table. Considering the average of the global sample size error for α between 0.1 and 0.9, the next candidates for $t_\star$ are: *Trans* with 4.6% sample size error and 1,060,820 related tuples, *Disposition* with 9.5% and 16,130 related tuples, and *District* with 10.3% and 9,946 related tuples. Results confirm the strong relation between the sample size error and number of related tuples of the starting table.

As a conclusion, we can say that VFDS is very close to the best solution in terms of global sample size error.

3) Query relative error: We illustrate the queries used for the approximate query evaluation in table I. Query $Q1$ determines the average amount of a loan (*Loan*), $Q2$ the average payments value of accounts with loans of a given status, `'A'` (*Account*⋈*Loan*), $Q3$ the average amount value of orders of accounts from the district region `'west Bohemia'` (*Account*⋈*'Order'*⋈ *District*), $Q4$ the average balance value of transactions for cards of type `'classic'` (*Account*⋈*Trans*⋈*Disposition*⋈*Card*). Fig. 4 shows the average of the absolute value of the query relative error for JS, LBS, and VFDS over the 5 runs for the queries $Q1$, $Q2$, $Q3$, and $Q4$. The more data is inserted in the sample database, the more the sample answer will resemble the original answer. Thus, we observe a tendency for the query relative error to decrease as the sampling rate increases for all queries and all methods. As generally JS produces a sample database with a higher percentage of data, it will generally provide closer results to the original results of the queries for smaller values of α, producing a sample with a smaller variation for this error.

Fig. 4-a shows the results for $Q1$ relative error for JS, LBS, and VFDS. We observe that it varies between 5.2% and 0.9% for JS, between 6.1% and 0.5% for LBS, and between 7.5% and 0.9% for VFDS. We observe that LBS generally provides better responses to $Q1$ with an average query relative error of 2.5%, while JS and VFDS perform quite similar (2.8% and 3% respectively). Fig. 4-b shows the results for $Q2$ relative error for JS, LBS, and VFDS. We observe that JS varies between 9.4% and 0.6%, LBS between 11.5% and 0.8%, and VFDS between 10.6% and 1.3%. We notice that JS performs best with an average query relative error of 3.06%, while VFDS and LBS perform quite similar with 3.5% and 3.6% respectively. Fig. 4-c shows the results for $Q3$ relative error for JS, LBS, and VFDS. We observe that JS varies between 11.3% and 0.9%, LBS between 6.1% and 0.8%, and VFDS between 6.8% and 1.5%. We observe the average query relative error for JS is 4.15%, for LBS 3.2%, and for VFDS 3.55%. Fig. 4-d shows the results for $Q4$ relative error for JS, LBS, and VFDS. We notice that JS varies between 12.5% and 1.1%, LBS between 24.6% and 0.9%, and VFDS between 10.6% and 1.1%. We observe that LBS generally provides worst responses to $Q4$ with an average query relative error of 5.6%, while VFDS performs best with 3.5%, leaving JS in the middle with 5%.

As a conclusion, we can say that all approaches generally perform quite similar in terms of query relative error as they represent random sampling methods.

V. Conclusion

In this paper we proposed a database sampling approach targeting relational databases that aims to minimize the time necessary to produce a sample database. The main contribution of VFDS is to sample a database at high speed, given a sampling rate from the user. The system selects an appropriate starting table for the sampling method according to the target database. VFDS offers the possibility to the user to select a desired starting table. However, this generally requires expert knowledge about the database under analysis as the starting table critically impacts the sample database, and the automated sampling strategy is recommended. Random tuples are sampled from the starting table along with their associated tuples from the rest of the tables in the database. The associated tables are filled only with the tuples that are needed to satisfy foreign key constraints in order to minimize the space overhead of the sample database. VFDS drastically reduces the time needed to produce database samples (between 300 and 1,000 times faster) with respect to existing approaches by performing the sampling in a single pass over the entire database. Results also show that VFDS is precise in terms of sample size and approximate query answering.

As future work, we plan to propose a mechanism to deal with cycles of foreign key constraints in the original database. We plan to run additional experiments to demonstrate that our approach is relevant to other specific application areas such as data mining, and histogram construction. Last but not least, we plan to apply VFDS on a production environment available from our industrial partner IBM, use it as a testing environment, and observe significant reduction in execution time for producing a sample database, while maintaining the accuracy of the queries applied on the testing environment.

VI. Acknowledgments

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

References

[1] C. Binnig, D. Kossmann, and E. Lo, "Towards automatic test database generation," *IEEE Data Engineering Bulletin*, vol. 31, no. 1, pp. 28–35, 2008.

[2] T. Jones, "Software cost estimation in 2002 (cross talk)," June 2002.

[3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy, "Join synopses for approximate query answering," in *International Conference on Management of Data*, 1999, pp. 275–286.

[4] R. Gemulla, P. Rösch, and W. Lehner, "Linked bernoulli synopses: Sampling along foreign keys," in *International Conference on Scientific and Statistical Database Management*, 2008, pp. 6–23.

[5] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "An AGENDA for testing relational database applications," *Software Testing, Verification and Reliability*, vol. 14, no. 1, pp. 17–44, 2004.

[6] K. Taneja, Y. Zhang, and T. Xie, "MODA: Automated test generation for database applications via mock objects," in *IEEE/ACM International Conference on Automated Software Engineering*, 2010.

[7] C. Olston, S. Chopra, and U. Srivastava, "Generating example data for dataflow programs," in *International Conference on Management of Data*, 2009, pp. 245–256.

[8] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," *SIGMOD Record*, vol. 23, no. 2, pp. 243–252, 1994.

[9] "IBM DB2 Test Database Generator," http://www-306.ibm.com/software/data/db2imstools/db2tools/db2tdbg/.

[10] N. Bruno and S. Chaudhuri, "Flexible database generators," in *International Conference on Very Large Data Bases*, 2005, pp. 1097–1107.

[11] X. Wu, Y. Wang, S. Guo, and Y. Zheng, "Privacy preserving database generation for database application testing," *Fundamenta Informaticae*, vol. 78, no. 4, pp. 595–612, 2007.

[12] R. Yahalom, E. Shmueli, and T. Zrihen, "Constrained anonymization of production data: a constraint satisfaction problem approach," in *Workshop on Secure Data Management*, 2010, pp. 41–53.

[13] F. Olken, "Random sampling from databases," Ph.D. dissertation, University of California at Berkeley, 1993.

[14] G. John and P. Langley, "Static versus dynamic sampling for data mining," in *2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, 1996, pp. 367–370.

[15] J. Bisbal, J. Grimson, and D. Bell, "A formal framework for database sampling," *Information and Software Technology*, vol. 47, no. 12, pp. 819–828, 2005.

[16] X. Lu and S. Bressan, "Sampling connected induced subgraphs uniformly at random," in *International Conference on Scientific and Statistical Database Management*, 2012, pp. 195–212.