TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Software Science

Igor Volkov 143772IAPM

EXTENDING POSTGRESQL DATABASE MANAGEMENT SYSTEM TO ADD SUPPORT OF DATA MASKING

Master's thesis

Supervisor: Erki Eessaar
PhD

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Igor Volkov 143772IAPM

POSTGRESQL ANDMEBAASISÜSTEEMI LAIENDAMINE ANDMETE MASKIMISE VÕIMALUSTEGA

Magistritöö

Juhendaja: Erki Eessaar
Doktor

Tallinn 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Igor Volkov

15.05.2017

Abstract

Data masking is a difficultly reversible process that replaces sensitive and valuable data with data that looks realistic. However, in fact data is generated or otherwise computed and thus cannot be used for unauthorized purposes.

PostgreSQL is an advanced SQL database management system (DBMS) that as of May 2017 ranks as the second most popular open source DBMS and fourth most popular DMBS in general [1] . Yet there is an apparent lack of data masking solutions that are free, open source, and native to the DBMS.

This work explores the types and techniques for data masking described in the existing literature, implements some of the techniques in PostgreSQL as a proof-of-concept open source extension, and presents a performance test of the implemented solution in case of masking a simple database.

As the implemented extension is far from being perfect, its limitations, possibilities for the future development, and necessary refactorings are also presented.

The software is licensed with the MIT license. The source code is published at https://gitlab.com/thodt-md/themask/tree/devel .

This thesis is written in English and is 90 pages long, including 8 chapters, 9 figures figures and 11 tables.

Annotatsioon
PostgreSQL andmebaasisüsteemi laiendamine andmete maskimise võimalustega

Andmete maskimine on raskesti tagasipööratav protsess, mis asendab tundlikud ja väärtuslikud andmed realistlikuna näivate andmetena. Tegelikkuses on need andmed genereeritud või olemasolevate andmete põhjal arvutatud. Seega ei saa neid andmeid otstarbe (näiteks süsteemi testimine) väliselt kasutada. Leidub hulgaliselt erinevaid andmete maskimise algoritme. Neid realiseerivad mitmed andmebaasisüsteemid ja ka andmebaasisüsteemidest eraldiseisvad programmid.

PostgreSQL on võimekas SQL-andmebaasisüsteem (DBMS), mis 2017. aasta mai seisuga on populaarsuselt teine avatud lähtekoodiga andmebaasisüsteem ja populaarsuselt neljas üldises andmebaasisüsteemide populaarsuse pingereas [1] . Samas ei leidu selle jaoks andmete maskeerimislahendusi, mis on tasuta, avatud lähtekoodiga ja integreeritud andmebaasisüsteemi.

Antud töö annab kirjandusel põhineva ülevaate andmete maskeerimisvõimalustest, realiseerib osa võimalustest kontseptuaalse prototüübina (*proof-of-concept*) PostgreSQL avatud lähtekoodiga laiendusena ning mõõdab loodud laienduse abil tehtavate operatsioonide jõudlust lihtsa andmebaasi maskimise põhjal.

Kuna loodud laiendus on ideaalist kaugel, siis töös kirjeldatakse ka selle puuduseid, edasisi arendusvõimalusi ning refaktoreerimise vajadusi.

Jõudlustestide kohaselt ei muutnud maskeerimisoperatsioonid andmete kopeerimist märgatavalt aeglasemaks. Kõige aeglasemaks operatsiooniks osutus juhuslike isikunimede genereerimine. Selle kood vajab refaktoreerimist. Kuigi jõudlustestid näitavad, et prototüüp on töövõimeline, siis täiuslikuks andmete maskeerimise lahenduseks saamine nõuab lisatööd.

Tarkvara on litsenseeritud MIT litsentsiga. Tarkvara lähtekood on publitseeritud https://gitlab.com/thodt-md/themask/tree/devel.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 90 leheküljel, 8 peatükki, 9 joonist, 11 tabelit.

List of abbreviations and terms

AES	Advanced Encryption Standard – “a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001” [2] .
CSV	Comma-Separated Values – standardized textual format, which is commonly used for storing tabular data [3] .
CRUD	Create, Read, Update, Delete – basic data manipulation operations.
DBA	Database Administrator – a person responsible for database maintenance and operation [4] .
DBMS	Database Management System – “a set of programs that enables users to store, modify and extract data from a database” [5] .
DDL	Data Definition Language – a subset of a database domain-specific language, meant for altering the structure of the database. SQL operators that are used in its DDL statements include CREATE, ALTER and DROP.
DML	Data Manipulation Language – a subset of a database domain-specific language, meant for querying, persisting and modifying data in the database without altering its structure. SQL operators that are used in its DML statements include SELECT, INSERT, UPDATE, DELETE and MERGE.
FPE	Format-Preserving Encryption – an encryption technique that allows the encrypted text to be in the same domain as plaintext. For example, encrypting a valid credit card number will yield another valid credit card number [6] .
JSON	JavaScript Object Notation – lightweight textual data interchange format, based on a subset of the JavaScript programming language [7] .
JSONB	Binary data type that can be used in PostgreSQL for a more efficient JSON data storage [8] .
LOB	Large Object – a term for types of unstructured (in the sense that the DBMS cannot operate on its structure) binary (BLOB) and character (CLOB) data in some SQL DBMSs. The values that belong to the types have typically very high size limits and are stored in a location separate from other data in the database (outside of regular data blocks where data is stored row or column wise).

LUT	Lookup table – an array-like structure that is used to replace an expensive computation with a simpler array-lookup [9] .
PIC	Personal Identification Code (Estonian: *isikukood*) – a numeric code issued by an Estonian governmental agency. The code is formed on the basis of the gender and the date of birth of a natural person and allows the specific identification of the person [10] .
PL/pgSQL	Procedural Language/PostgreSQL - built-in loadable procedural language for the PostgreSQL DBMS that can be used to perform server-side database management and data processing [11] .
RDBMS	Relational Database Management System – a DBMS where the data is presented to the database users as values in the relational tables [12] . Internal storage of the data is not prescribed by the relational model.
SQL	Structured Query Language – a standardized concrete (as opposite to the abstract language like a data model) language for managing databases [13] that has been created according to the underlying data model of SQL. The model is similar to but not equivalent with the relational data model. SQL is a database domain-specific language.
SQL DBMS	SQL Database Management System – database management system that allows organization of data according to the underlying data model of SQL and offers SQL database language to work with the data.
SQL proxy	SQL proxy is a type of a computer system or a software application that forwards SQL requests from a SQL DBMS client to the SQL DBMS server and the related responses back to the client. The proxy can cache, alter, or drop the incoming requests.
TUT	Tallinn University of Technology
UML	Unified Modeling Language – standardized general-purpose graphical modeling language that can be used to analyze, specify, and document the artifacts in an information-intensive system [14] .
XML	Extensible Markup Language – textual data interchange format recommended by W3C [15] .

Table of Contents

List of Figures

List of Tables

1 Introduction

Large complex projects, especially in the public sector tend to fail [16] [17] . Typically such projects are commissioned as large systems of undifferentiated high and low-risk components. These components often rely on proprietary technologies that are left from legacy systems. Such components may be deeply integrated with the supporting logic of the underlying platforms. Thus, one has to deal with these only as whole [18] . This leads to the limited possibilities to change the system. It raises budget costs, delays deployment, and possibly ends with total project failure. As a consequence, software development tends to move towards doing faster release cycles, designing loosely coupled components, using leaner processes and open source solutions.

As an example, while working for a consultancy firm during the last five years, I have noticed a clear trend of moving away from proprietary DBMSs like Oracle towards free and open source solutions, PostgreSQL being the primary choice for all new projects requiring a SQL DBMS.

However, developing integrated applications of any scale requires correctly functioning services that the application can use. During the development the services should be implemented either as mocks, which are hard to maintain and really feasible only for the simplest components, or real services with non-production data.

Most of the clients I have worked with more application environments, than just “development”, “test”, and “production”. Other environments may include, for example, “training”, where a production artifact is paired with production-like test data and is used to train employees.

Since the development cycle is shorter, the software is more likely to suffer from bugs and unexpected behavior due to data in production being dissimilar to the data used in development and during testing. The usual production issue resolving process done for a client involves the developer, to whom the issue is assigned, systems or database administrator (DBA), who has access to the production environment. In addition, 2-6 people besides these are involved. For solving an obvious issue, obfuscated production log excerpts may suffice. However, now and then a hardly-reproducible bug emerges and the developer has to either debug on-site or blindly ask questions in an issue tracker. After all other configuration and environment-related factors are excluded, the issue may turn out to be input-related and thus could have been more easily resolved, say, with an obfuscated production copy.

There are likely many ways to solve these issues, but one of them, requiring perhaps least change, is data masking. Data masking is a difficultly reversible process that replaces sensitive and valuable data with data that looks realistic. However, in fact the data is generated or otherwise computed and thus cannot be used for unauthorized purposes.

The need and the apparent lack of a data masking solution for PostgreSQL that is:

- ***free***, so that the client bears no additional expenses to provide production-like test data,
- ***open source***, opening the possibility for code review and modification,
- ***independent***, i.e. requires no third-party software dependencies besides PostgreSQL

has driven me to implement such a solution.

To achieve this goal the thesis firstly intends to provide a theoretical basis for data masking – explore the types and techniques for data masking described in existing literature, as well as mentioning some of the existing implementations.

As there are commercial proprietary solutions for data masking from DBMS and independent software vendors, the thesis will give a short overview of their capabilities.

The solution is to be implemented using the extension mechanism of PostgreSQL. The mechanism of action of the solution, implemented masking techniques, and the peculiarities of packaging the related objects into an extension will be discussed. Known limitations and refactoring possibilities of the implementation will be mentioned.

To demonstrate the usage and to test the performance of the extension a simple database will be masked by using the extension.

Finally the scope of the future work and the summary will be presented.

As such the work is an example of Design Science research that should end up with an artifact (in this case a proof-of-concept data masking solution for PostgreSQL) [19] . Development of the artifact uses business requirements and existing knowledge from the field as an input. It produces an artifact and evaluates it. The results of the evaluation are inputs to the further development of the artifact.

2 Data Masking

There are many definitions of data masking, but all of them seem to converge on the notion that data masking is a process that replaces sensitive and valuable data with data that looks realistic, but is in fact generated or otherwise computed and thus cannot be used for unauthorized purposes [20] [21] [22] (p. 2)[23] (p. 2). Masked data is typically needed for untrusted non-production environments. Proper data masking should not be easily reversible [24] . Data masking is relevant regardless of the underlying data model (relational, SQL, object-oriented, graph-based, document-based, etc.) of the database. However, because the thesis develops a data masking solution for a SQL DBMS the following sections refer specifically to the elements of the SQL data model (tables and columns).

2.1 The Need for Data Masking

While some may argue that obtaining masked data can be achieved by using simpler means, such as using a hand-crafted data manipulation language (DML) script, industry leaders and independent security experts warn against using such techniques [23] (p. 5) [24] because this approach is:

- ***not reusable***: DML script (a set of DML statements) taken from one database will only be reusable if the other database has a very similar structure; Moreover, different SQL DBMSs have different SQL dialects and thus such statements may need rewriting if moved between different DBMSs.
- ***not maintainable***: every now and then applications are upgraded, adding new columns with sensitive data to the schema, renaming tables or columns, or removing tables or columns, which may not have necessarily contained sensitive data. However, in each case each script must be carefully revised to accommodate new changes. It lengthens the upgrade development cycle.

In addition to that, it should be also pointed out that a single script solution is not flexible and will hardly be reusable, if the data is to be masked with different rules for different scenarios even for the same database (such as described in [23] , p. 4).

Another common misconception is that production data masking can be replaced with random test data generators. One can implement it as case-based set of scripts, implement the more general software themselves, or use some existing software (like [25] for that). While this may be true for a limited number of cases, say for development data, configuring the random data generator will require the same, if not greater effort as configuring a data masking solution.

1. Production database needs to be analyzed and reviewed in respect to what data is contained in each column, both sensitive and not sensitive.

2. Relations between columns need to be considered and maintained in the generated data, both explicit (uniqueness/foreign key constraints) and implicit (denormalized data in tables, meaning that the same facts are repeated in multiple places in the same table or across tables),

3. An appropriate data generation method must be chosen for each column.

4. Generated data must be reviewed and the methods appropriately adjusted, if necessary.

Similar steps are described in F.A.S.T. data masking approach by Oracle [23] (p. 5). The acronym refers to the Find, Assess, Secure and Test steps of data masking.

In addition to that, random test data does not help in any way during the resolution of production issue. If a statistical or medical research is to be done on the production data it requires some of the data that is essential to the research to be real facts. On the other hand, data that allows scientists to identify persons and their concrete problems, must most probably be obfuscated.

If it is in fact decided to use data masking for a particular problem, then there are different techniques and architectures to choose from.

2.2 Data Masking Architectures

There are mainly three distinct types of masking architectures described – static, dynamic, and on-the-fly. The classification is made in respect to where the sensitive data is stored, when the data is masked, and how it is accessed [20] [21] [22] . Brief description of each, including its advantages and disadvantages is given below.

2.2.1 Static Data Masking

Static data masking architecture is illustrated in Figure 1.

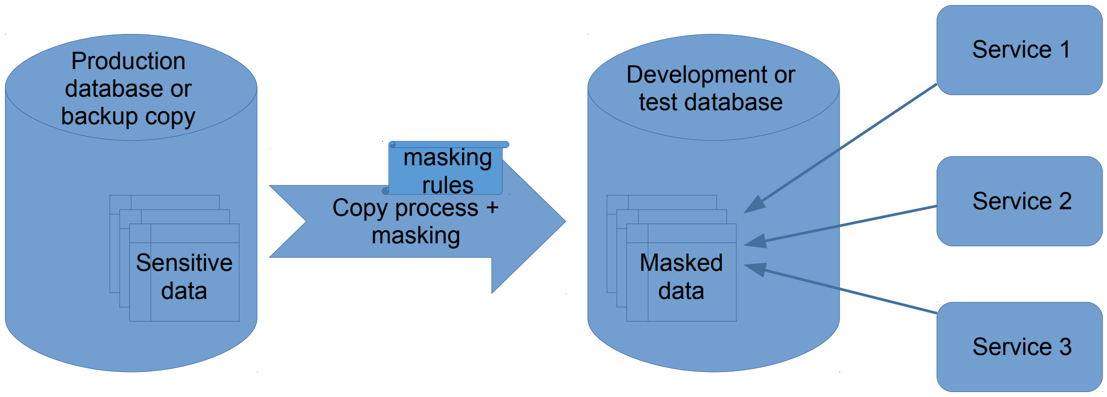


Figure 1: Static data masking architecture

With static data masking sensitive data is masked while it is being copied to a non-production database. It is a one-time process that produces a static copy with masked data. Client applications that access the masked database get the same view regardless where they reside and what they are otherwise authorized to do with the database.

Notable advantages of using this architecture include the following.

- ***Security***: since sensitive data does not leave the production database, an attacker will be able to access sensitive data only if he/she has direct access to the production database. Traditional security means, such as networking firewalls, physical separation, and security functionality of DBMSs can be deployed.
- ***Performance***: queries on a static database with masked data will arguably be faster, since no additional computation is done with the data to mask it while executing the query. For example, if a column is dynamically masked by using a function in a view, then the function will be executed each time the column is

accessed. It will be done so many times as there are rows in the query result. To speed up the query, one has to create a function-based index to the column.

However there are also disadvantages.

- Errors occurring during the masking process may effectively halt it. Restarting the work means doing the process from scratch.
- Inability to change the masking rules after the process has finished, since they would not have any effect on the data already masked. Thus, to apply new masking rules the entire copy has to be produced again.
- Since the data is being masked while copying by an external or built-in tool, this may interfere with using alternative replication solutions.

2.2.2 Dynamic Data Masking

Dynamic data masking architecture can be graphically described by using Figure 2.

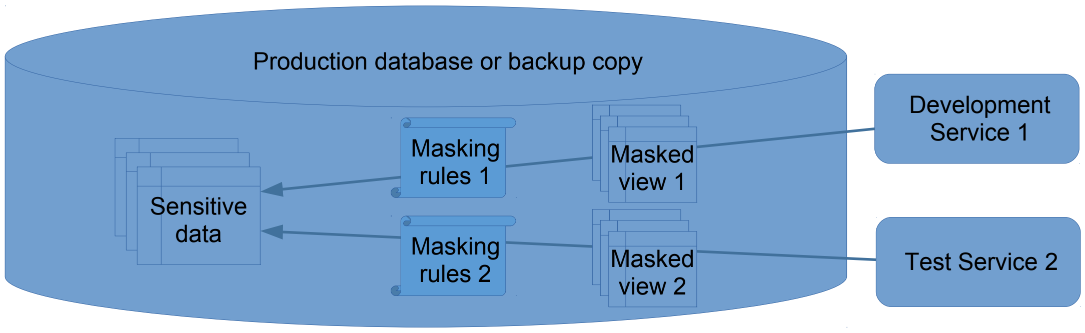


Figure 2: Dynamic data masking architecture

With dynamic data masking sensitive data resides in the production database or is copied (without masking) to another database. Different masking rules are then applied based on the requesting source, user identity, etc. In effect the masking operations are done on-the-fly as the database clients make requests and the data is not duplicated for each set of rules.

Dynamic data masking has the following advantages.

- ***Storage***: less storage is needed to host the data, because there is no separate copy of the masked data.

- ***Flexibility***: rules may be added and removed at any time, the clients will see changes immediately.

Disadvantages include the following:

- Depending on the implementation, queries that access the data have to be rewritten in order to use masking functions in the SELECT clause and possibly in the WHERE and HAVING clauses as well. Alternatively, base tables should be renamed and views that have the same name as the original base tables should be created. The views invoke masking functions. If different parties should have data masked based on different rules, then the latter approach is problematic, because there cannot be multiple views with the same name in the same schema. Moreover, data modification through views has big limitations in SQL databases, leaving some data modification functionality unusable. Thus, realistically, one has to modify applications/services that access the data to make them function with dynamically masked data as if it is normal data.

- High risk of data being compromised. If an adversary gains privileges either by stealing production credentials, IP address spoofing or via some sort of privilege escalation attack, then sensitive data can be fully recovered.

- Depending on the implementation, not all DBMS features may be usable. For example, if the solution is implemented using a form of SQL proxy that rewrites all incoming requests, features such as stored routines may uncover the masked data [26] . The proxy also needs to be up to date in respect to the target DBMS version to support new language features. Otherwise it may render masking ineffective.

- In case of materialized views (snapshots) the dynamic masking that is applied in case of accessing base tables should be duplicated or the view creation process should produce data that is already masked. In either case the duplication of masking rules makes it more difficult to manage these. The latter is an example of static data masking and shows that a data masking solution may combine different strategies.

- Not all data masking techniques may effectively be used. For example, if the column is to be randomly shuffled, then the data will have to be reshuffled after adding any records. Otherwise data in new records will maintain its insertion order (the purpose of shuffling is to sever the link between the shuffled value and its original record).

- A condition in a query based on data that needs masking might be something like that:
  - `WHERE f_masking_function(column_name) = something`
  - If the data that needs masking is in an indexed column, then one may want to create a function-based index to the column that the DBMS can use in case of searching from masked data. If for different parties the masking function for the column is different, then for each party there should be a separate index. Indexes increase data size and reduce the performance of update operations.

2.2.3 On-the-Fly Data Masking

There is also a third architecture advertised by some vendors [27] that involves a form of data replication. It is similar to the static architecture in the sense that data does not leave the production database unmasked. However, data modifications are regularly propagated into a masked copy instead of just creating copy once and then using it. Masking rules can be changed, but the changes will not propagate to the data that is already masked. This may be an area of concern if the rules for a column have changed in such a way that it is obvious by comparing older an newer records. However, when applied correctly, this allows us to have a more up-to-date masked copy of production data.

2.3 Data Masking Techniques

There are many techniques available [22] [28] [29] that can be used to protect sensitive data, with new ones appearing in response to new demands and technological advancements [30] . Below is a short description of frequently used techniques:

2.3.1 Variable Suppression

It is also known as “Nulling Out”. This method involves replacing a value with `NULL` or a constant (such as “REDACTED” or “***”) that is not related with the data being masked.

It is the simplest data masking method.

2.3.2 Truncation or Cropping

This method involves removing part of a string value or decreasing precision of a numeric or a time-stamp value. For example, if it is necessary to know the year a car has been purchased, then the month and day of that year are irrelevant. Thus, for instance, a date 05.04.2017 can be truncated with the precision of month or year to 01.04.2017 or 01.01.2017, respectively.

2.3.3 Substitution

This method replaces a value with another similarly looking precomputed random value. The replacement may or may not be related to the value being replaced. For example, real full names in a column are replaced with random full names. The new names are not just random string but are randomly selected from a set of precomputed human names.

The main disadvantage with using substitution is the necessity to have an adequately sized pool of random values. If the names of a pair of persons are different in the original data, then these should be different in the masked data as well. Moreover, the process has to be consistent (i.e. name John Smith is replaced with Bob White in all contexts) [28] . The more data duplication within or across tables there is, the more work it takes to ensure the consistency.

2.3.4 Shuffling

Shuffling is similar to Substitution. The precomputed value pool consists of the real values in the masked column, and the values are randomly swapped between rows. The main advantage is that the data is kept real, but any link to the original row is severed.

This allows the shuffled values to retain summary statistics, such as mean and standard deviation.

However, shuffling may be ineffective for a small number of observations and in case an adversary wants to know only if a value is present in the column [20] (p. 537).

2.3.5 Masking Out

It involves replacing characters of an unmasked value with a static masking character, such as "*" or "X". Commonly used for textual data, such as addresses, emails, account numbers, credit card numbers, and phone numbers.

2.3.6 Random Noise

It is also referred to as "Numeric and date variance", "Blurring", and "Skewing". This technique is used mainly on numeric and temporal data, when knowing the precise value is not necessary for it to be useful. Examples of application may include date of birth, invoice totals, etc.

Random noise technique computes a random fraction (within predefined limits, say 10% of a numeric value or 100 days for a date value) of the value and then adds it to or subtracts it from the original value.

2.3.7 Encryption

It is an encryption technique as the name suggests. It encrypts the data using a strong cypher like Advanced Encryption Standard (AES). As a result, the value, say ASCII character data, may fall out of its original domain and become binary data, with a side-effect of increased data size.

2.3.8 Format-preserving Encryption

Format-preserving Encryption (FPE) is a special case of the Encryption technique. It does not leave the data scrambled and thus unusable. Cyphertext is in the same domain as plaintext [6] [29] . Encrypting a credit card number in a format-preserving manner will deterministically produce another credit card number, a social security number yields another social security number, etc.

The advantage of this method is obvious – it can be as secure as Substitution technique with an unlimited precomputed data set and as fast as Encryption.

However, FPE must be implemented for each domain separately, depending on its message space. Data such as credit card numbers, social security numbers as well as other national identification codes often include a form of checksum, which is a derived part of the plaintext value and must be recomputed after the encryption.

2.3.9 Methods Based on Linear Models

Besides the already mentioned techniques, complex Perturbation-based methods exist for masking multi-column numeric data. The methods involve building a linear model of the data being masked and using it to add noise to each observation, so that the overall resulting masked data has the same mean vector and covariance matrix as the original data [31] [32] . While it is an interesting approach, more detailed overview is beyond the scope of this thesis.

2.4 Risks and Challenges in Data Masking

There are risks and challenges when using any data masking solution. Some worth mentioning are presented in the next sections.

While this work acknowledges the aforementioned risks and challenges, attempting to solve them is not in the focus of this thesis. The thesis only provides an artifact for data masking and it is up to the user to decide what masking rules to apply for a particular purpose. Any organization may have its own measure of what is valuable data, which may be more strict than the law and operating regulations require.

2.4.1 Risk of Accidental Disclosure

If the data to be masked is not carefully analyzed in respect to correlation between unmasked values, then a re-identification attack can be mounted using common data mining techniques [33] .

Some security researchers advise using only such masking methods that can guarantee a low risk of re-identification. For example, previously mentioned Variable Suppression, Substitution and Shuffling would be good choices in this regard. It is argued that simply having masked some data may not be enough, if re-identification risk is not measured [34] .

It should also be noted that sensitive data disclosure not only leads to a reputation loss, but may have legal consequences in most countries as well, including Estonia [35] .

2.4.2 Masking Synchronization

When masking a row that for example contains full name and gender columns, chosen masking methods must be in sync with each other: a random name must be of the same gender, if they are both randomized.

Another good example is the Estonian Personal Identification Code (PIC): it encodes gender and date of birth [36] . If it is to be randomized, special care must be taken to appropriately mask the date of birth column and person name columns, if present.

2.4.3 Risk of Data Being Unusable

Many research papers have been published on the topics of k-anonymity and disclosure risk *vs.* data utility [37] [38] [39] [40] . One of the messages they convey is that there is a trade-off between data utility and disclosure risk. Data utility is a "measure of the business value attributed to data within specific usage contexts" [41] . By increasing the number of unmasked variables, which may be so called *quasi-identifiers*, we also increase the risk of disclosure. However, if we suppress all variables, data will not be useful and data utility decreases. A careful balance between acceptable disclosure risk and data utility should be obtained.

2.4.4 Masking Values that Belong to Structured and Large Object (LOB) Data Types

Many SQL DBMSs support storing data that belongs to the structured textual data types, such as XML and JSON and unstructured large object types, such as CLOB and BLOB (PostgreSQL analogues of the latter are TEXT and BYTEA [42] , respectively). XML, CLOB, and BLOB data types are prescribed by the SQL standard [13] .

Implementing an automated masking solution for structured document types implies that their values should have the same schema. It may not always be true, for example, in a database that stores raw requests and responses for a web service audit log.

Unstructured large objects may be user comments, where they may disclose sensitive personal information, or binary objects such as photos and videos. Generating fake data that could closely resemble such data, is infeasible and masking technique options are typically limited to the simplest ones, such as nulling out.

2.4.5 Data Integrity

The masking methods may break data integrity, when the column is part of a foreign key or a unique constraint, or has a CHECK constraint or is a part of a foreign key. Variable suppression can violate NOT NULL constraints. The masking methods can also break integrity checks that are implemented procedurally in triggers.

2.4.6 Misconfiguration

Any software product can be misconfigured either by accident or intentionally. While software vendors (and I myself) do their best to safeguard the end-user against wrong configuration options, this is not always possible. A test run with smaller data set must therefore be done to make sure everything is working as intended.

3 Existing Data Masking Implementations

Data masking in SQL databases has a well established market with many vendors offering solutions of different complexity. In this part an overview of built-in data masking solutions from SQL DBMS vendors, as well as external tools targeted for SQL DBMSs from independent software vendors will be given. The overview is based on white papers and manuals released by the vendors.

Search was done by using Google Search with the terms “postgres data masking”, “data masking sql”, “postgres data masking extension”, “built-in data masking database”, “open source data masking sql”, “data masking database github”, “data masking oracle”, “data masking gartner”, “data masking sqlserver”, and possibly a few others I have currently no recollection of.

Search was also done on the popular site PostgreSQL Extension Network where the authors of PostgreSQL extensions publish their work [43] .

3.1 SQL DBMS Implementations

First of all, there are SQL DBMSs that have built-in data masking facilities. Their comparison is in Table 1.

Table 1: Built-in masking facilities of some SQL DBMS

Vendor	**DBMS or package**	**License**	**Type**	**Implementation details**
Microsoft	SQL Server 2016 [44]	Proprietary	Dynamic	An SQL extension. Data masking is applied column-wise using `MASKED WITH (FUNCTION = '…')` syntax. Four built-in techniques are available: Default, Email, Random, and Custom String.
IBM	DB2 11 [45]	Proprietary	Dynamic	An SQL extension. Data masking is

Vendor	DBMS or package	License	Type	Implementation details
				applied column-wise using `CREATE MASK` statement. Masking itself is done using a form of `CASE` statement. No mentioning of any built-in masking techniques.
Oracle	Oracle Database 12c – Data Masking and Subsetting Pack [46]	Proprietary	Static	A graphical web-based utility that lets the user define masking rules using a variety of built-in masking techniques. A masking script is then generated. The script can then be scheduled to be executed periodically. It has seven implemented techniques (Shuffle Masking, Blurring or perturbation, Encryption, Format Preserving Randomization, Conditional Masking, Compound Masking, and Deterministic Masking). The option for a user-implemented masking method is available.
Oracle	Oracle Database 12c – Data Redaction Pack [47]	Proprietary	Dynamic	Rewrites query results. Masking policies are configured using the routines in the `DBMS_REDACT` package. Four masking methods are available: Full redaction, Partial reduction, Regular expressions and Random redaction.
Fujitsu Software	Enterprise Postgres [48]	Proprietary	Dynamic	A closed-source proprietary PostgreSQL version with the data masking support. No mentioning how exactly it can be configured, but a variety of masking techniques are available: "character shuffling, nulling or deletion, encryption, masking, and word substitution." Masking is implemented by rewriting query results.

The most feature-rich solution among DBMSs seems to be Data Masking and Subsetting Pack for Oracle Database 12c. However, it must also be noted that a custom PostgreSQL flavor by Fujitsu Software has data masking implemented as well.

3.2 Implementations Done by Independent Software Vendors

Secondly, there is a number of data masking solutions, external to DBMSs, done by independent software vendors. An overview of them can be seen in Table 2.

Table 2: Masking solutions by independent software vendors

Vendor and product	**License**	**Type**	**Postgre-SQL support**	**Implementation details**
DataSunrise Database Security Suite 3.7.1 [49]	Proprietary	Static, Dynamic	Yes	Dynamic masking implemented as a SQL proxy. Supports 26 different masking operations.
Delphix Data Masking Engine 5.1.6 [50]	Proprietary	Dynamic	Yes	An in-memory dynamic solution that offers eight data masking algorithms: Secure Lookup Algorithm, Segmented Mapping Algorithm, Mapping Algorithm, Binary Lookup Algorithm, Tokenization Algorithm, Min Max Algorithm, Data Cleansing Algorithm, Free Text Algorithm.
Camouflage CX-Mask [51]	Proprietary	Static	Yes	Supports over 20 data masking techniques.
Net2000Ltd DataMasker [52]	Proprietary	Static	No	A program with a graphical user interface that connects to target databases and executes masking operations. The solution implements the majority of masking methods written in whitepaper [22] , since the latter was written by the developers of DataMasker.
IBM InfoSphere Optim Data Privacy [53]	Proprietary	Static, Dynamic	Yes	Supports a variety of masking techniques, including FPE. Has an Eclipse Workbench-based user interface.

Vendor and product	**License**	**Type**	**Postgre-SQL support**	**Implementation details**
IBM InfoSphere DataStage Pack for Data Masking	Proprietary	Static / On-the-fly (?)[1]	Yes	Built on top of IBM DataStage, a fast ETL (Extract Transform Load – a term by IBM) tool. Feature wise seems to be similar to InfoSphere Optim Data Privacy [54] , but is designed to perform faster.
Informatica Persistent Data Masking [55]	Proprietary	Static / On-the-fly (?)[1]	N/A	Supports a variety of techniques: "substitution, blurring, sequential, randomization, nullification, plus special techniques for credit card numbers, SSNs, account numbers, and financial data." Yet, there is no explicit mentioning of PostgreSQL support.
Informatica Dynamic Data Masking [56]	Proprietary	Dynamic	N/A	SQL proxy with optional bypass. It is not explicitly stated that the solution supports PostgreSQL.
IRI FieldShield [57]	Proprietary	Static, Dynamic	Yes	An Eclipse Workbench-based graphical user interface, where the user can define masking rules and execute them. 12 built-in techniques for static data masking, including FPE. Dynamic data masking is available for C, .NET, and Java applications.
pgdump-obfuscator [58]	Open source	Static	Yes	A command-line utility written in Go programming language. Operates directly on `pg_dump` database backups. Supports some sort of email, password, and phone obfuscation.

The list of available solutions is in fact overwhelming, yet only a single open source utility was found that works with PostgreSQL. Many vendors provide no technical documentation, only marketing whitepapers. After doing this search I concluded that there are no available open source masking solutions for PostgreSQL.

[1] The exact implementation was not readily apparent after reading the manual.

Overall the overview of the products given here correlates with findings from other sources [59] .

Some research on data masking has been done in Tallinn University of Technology (TUT) before [60] . The author of that research compared various built-in data masking options as well, but part of his findings may be obsolete: the latest Microsoft SQL Server DBMS offers a more robust data masking solution.

4 Implementation Alternatives in PostgreSQL

As seen in the previous section, there are various options to implement data masking for a SQL DBMS.

- An external tool that works as a proxy and masks data dynamically.
- An external tool that masks data as part of database cloning process (on-the-fly or static masking).
- Built-in facility that masks data dynamically.
- Built-in facility that masks data as a part of database cloning/copying process (on-the-fly or static masking).

Using any external tool means extra resource in maintaining the tool, in addition to the DBMS itself, so one can imagine why DBAs would probably want to use a built-in facility.

4.1 PostgreSQL Extension Mechanism

PostgreSQL has a built-in facility to extend its functionality, possibility to create extensions, without rewriting its engine. An extension is a set of related database objects, such as tables, views, functions, types, etc. that are collected together as a single package [61] .

While it is possible to distribute related objects as a script, packaging them as an extension gives the benefit that PostgreSQL will recognize them as a single unit, which may easily be installed in a database using `CREATE EXTENSION` *`extension_name`* statement and just as easily removed by using `DROP EXTENSION` *`extension_name`* statement. Both statements are a PostgreSQL extension to the SQL standard.

Extension objects with the exception of configuration tables (if they are explicitly marked as such by the extension author) will not be a part of the backup.

Each extension package must consist of at least two files.

1. So called *extension control file* that must be placed in the installation's `SHAREDIR`[1]`/extension` directory. The file name must follow the pattern ***`extension_name`*****`.control`**.
   Parameters inside control files must follow the same convention as `postgresql.conf` file (i.e. `parameter_name = parameter_value`). Notable configuration parameters are listed in Table 3.

2. Extension SQL script with all the required DDL and DML operations. The script file name must follow the naming pattern ***`extension_name--version.sql`***. Extension scripts may have any SQL commands, except transaction control statements (for instance, `BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`) and commands that cannot be executed in a transaction block (for instance, `VACUUM`).

An extension can be iteratively developed and upgraded on existing databases by using so called *update paths* for SQL files. Update path file names must follow the naming pattern ***`extension_name--old_version--target_version`*****`.sql`**.

Control files for the newer versions can override parameter values of the old versions. Their naming pattern must be ***`extension_name--version`*****`.control`**.

Table 3. PostgreSQL extension control file parameters

Name and Type	**Description**	**Sample Value**
`comment` `(string)`	A comment about the extension, e.g. a short description.	`'An extension for new user type that allows X, Y and Z'`
`default_version` `(string)`	The extension version to install when not explicitly defined in a `CREATE EXTENSION` statement.	`'1.0'`
`requires`	Comma-separated names of the	`'plpgsql, pgcrypto'`

[1] An environment variable that depends on the distribution and operating system, in case of Debian Jessie this equals to `/usr/share/postgresql/9.6/extension` for PostgreSQL version 9.6.

Name and Type	Description	Sample Value
`(string)`	extensions that must be installed before this one.	
`relocatable (boolean)`	Can the extension be moved to another schema after installation?	`true, false`
`schema (string)`	Schema name for non-relocatable extensions. Required if an extension internally assumes that its related objects reside in some specific schema.	`'myextension'`

5 Designing and Implementing the PostgreSQL Extension

This chapter highlights the design of the implemented solution including functional requirements in terms of the main use cases and domain model. A brief overview of what are the main processes in case of using the extension are given. An overview of used patterns and best practices is made. Finally, it is explained how to install the extension and what are its known limitations.

5.1 On Using UML for Modeling PostgreSQL Extensions

While it is a fact that PostgreSQL supports so called *composite types*, which may have fields of various data types, and which can in turn be declared as a table column type, their usage beyond data transfer and storage is rather limited [62] . A declaration of composite type cannot have constraints, such as `NOT NULL`. Instead of bundling data (attributes) and functionality (methods) together into one whole like it is done in case of object-oriented programming, one has to separately define operators for performing operations with the data that belongs to the type.

In terms of object-oriented features PostgreSQL also supports table inheritance [63] and *subtyping* in case of range types [64] . I have no use for these features and decided not to use them to create the extension.

To date the only loadable stored procedure languages bundled with every PostgreSQL installation are SQL and PL/pgSQL [11] , which is a procedural language . Functions written in the latter are a set of SQL DML statements that are put together under a function name and can be thus invoked with one command. There are extensions for other languages, including object-oriented languages. However, they require separate installation together with the corresponding runtime. I believe that this is not portable and should be avoided.

Thus, expressing mechanism of action of an extension by using UML constructs meant for object-oriented scenarios, such as sequence diagrams, without diving into lower-level abstraction details of the DBMS (like the ones listed in [65]), can be challenging.

That said, one can still use other UML diagram types like use case diagram, activity diagram, and class diagram for describing an extension. The latter can be used for domain modeling as well as for describing the tables and views that are created in a database as the result of installing the extension.

5.2 Functional Requirements

The implemented extension is using static masking architecture. I believe that disadvantages of dynamic masking outweigh the advantages. In my experience, no organization would grant access to production DBMS servers anyway, regardless of the purpose.

Functional requirements of the extension are expressed by using use case diagram in Figure 3 and the short high-level descriptions of the use cases that follow the diagram. Characterization of use cases as secondary or primary are based on my subjective evaluation. All the use cases have been actually implemented in the extension.

Not all masking techniques described in part 2.3 were chosen. Some of them like FPE and methods based on linear models deal with heavy computation. Implementing them in SQL or PL/pgSQL would be problematic, possibly even detrimental to their performance [66] .

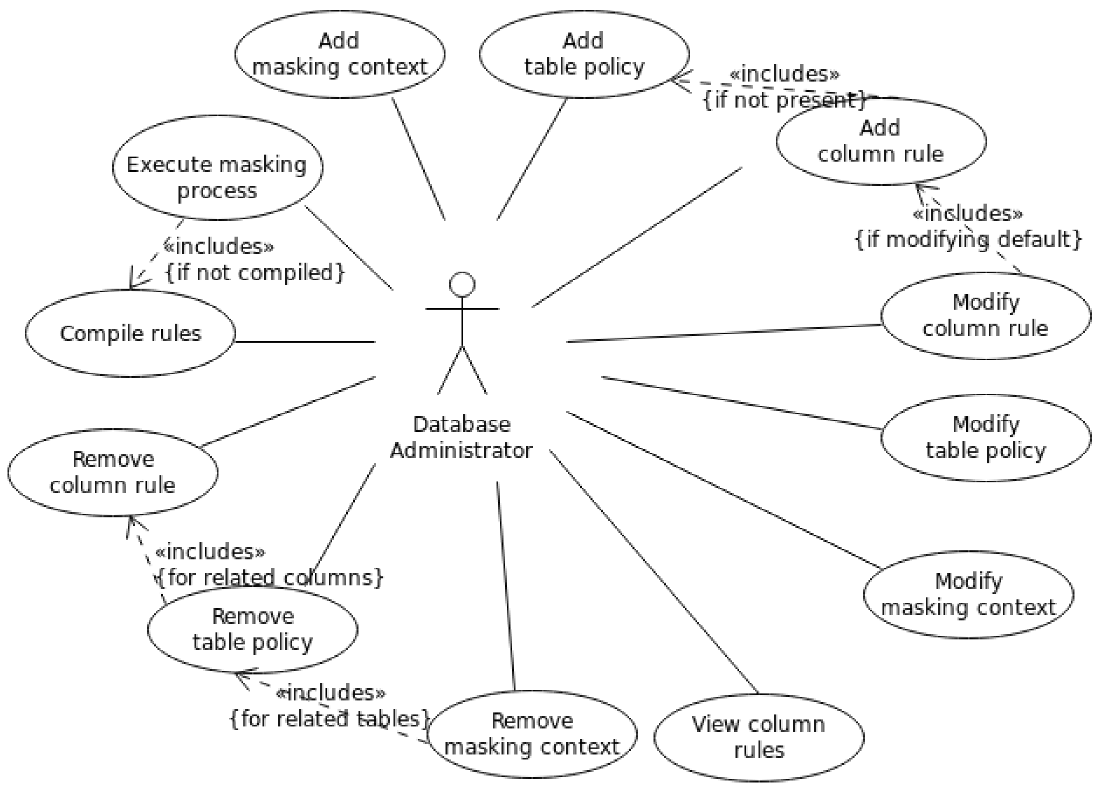


Figure 3: Extension use case diagram

A short description for each use case is given below.

5.2.1 Add Masking Context

Type: secondary

Actor: Database Administrator

Description: An actor defines a new masking context. A masking context has a name and contains global settings for a group of masked tables. Different masking operations on the same table can be configured in different masking contexts.

5.2.2 Modify Masking Context

Type: secondary

Actor: Database Administrator

Description: An actor modifies settings for an existing masking context.

5.2.3 Remove Masking Context

Type: secondary

Actor: Database Administrator

Description: An actor removes an existing masking context. This in turn removes all related table policies, column rules, and compiled operations.

5.2.4 Add Table Policy

Type: secondary

Actor: Database Administrator

Description: An actor adds a table policy to a masking context that defines default settings for a table to be masked. If the referred table does not exist it should produce an error.

5.2.5 Modify Table Policy

Type: secondary

Actor: Database Administrator

Description: An actor modifies an existing table policy for a masking context. If the referred table does not exist it should produce an error.

5.2.6 Remove Table Policy

Type: secondary

Actor: Database Administrator

Description: An actor removes an existing table policy for a masking context. This in turn removes all related column rules.

5.2.7 Add Column Rule

Type: secondary

Actor: Database Administrator

Description: An actor adds a masking rule for a table column. If a policy does not exist for the table, a default one is added. Column rule defines the masking operation to be performed on the column and its arguments, if necessary. The following masking techniques must be supported: Variable Suppression, Truncation or Cropping, Substitution, Shuffling, Masking Out, and Random Noise. If the referred column does not exist it should produce an error.

5.2.8 Modify Column Rule

Type: secondary

Actor: Database Administrator

Description: An actor modifies a masking rule for a table column. If the rule is derived from the table policy and does not exist by itself, a new rule is added explicitly. If the referred column does not exist it should produce an error.

5.2.9 Remove Column Rule

Type: secondary

Actor: Database Administrator

Description: An actor removes a masking rule for a table column. After the rule is removed the rule defined by the table policy must be used.

5.2.10 View Column Rules

Type: secondary

Actor: Database Administrator

Description: An actor views a list of all effective column rules, implicit (from the table policy) and explicit (added manually).

5.2.11 Compile Rules

Type: primary

Actor: Database Administrator

Description: An actor invokes rule compilation for a given masking context. As a result of the compilation, the added rules are checked for errors and a list of compiled operations is persisted. If there rules are not valid, an error is shown and the process aborts.

5.2.12 Execute Masking Process

Type: primary

Actor: Database Administrator

Description: An actor executes the masking process for a given context. As a result masked table copies are created in cloned schemas and filled with masked data. If the list of the compiled rules for the given context is empty, then the compilation process is executed beforehand (see 5.2.11). If there are no rules defined, then the process aborts. If there is no cloned schema, then a schema is created. If the cloned schema already contains tables, then the user has to explicitly allow table truncation, or else an error is reported.

5.3 Non-functional Requirements

There are some constraints to how the extension is to be implemented.

- **Open source.** Does the open source nature of the masking solution make it less secure? I believe that simply knowing the masking algorithm makes no difference on the probability of unmasking the data. The implemented algorithms should rely on using functions that are a part of the standard library provided by PostgreSQL such as `random()`. As long as the used functions are secure, the implementation is secure. I also believe that any software dealing with data masking and encryption should be open source, so that others may find possible flaws in its implementation. I encourage any interested party to review the code of the extension to prove or disprove the safety of the implemented extension.

- **No external dependencies.** The implemented extension must not use other extensions that require separate installation.
- **Platform independence.** The extension must be implemented using languages that do not require recompilation on every target platform.
- **No direct DML on the configuration tables.** The user must be able to configure the extension without using `INSERT`, `UPDATE`, or `DELETE` statements on the configuration tables. As an exception, DML is allowed on the table containing popular names as a part of the extension installation process. Generally, the configuration must take place by invoking functions.
- **Works on PostgreSQL 9.6+.** The extension should work on PostgreSQL version 9.6 or later. Working on lower versions is possible but not mandatory.
- **Published with MIT License.** Why the MIT license? There are other open source licenses available [67] . Some of them are more permissive (meaning they place less restrictions on the code reuse), some of them are so called *copy-left* licenses. Copy-left licenses are more restrictive in the sense that they may require to publish any changes of the code being used, as well as interfere with the license of the product the code is being used in. This may be undesirable for closed-source systems. MIT License text is rather short, easy to understand, and requires the user only to include the copy of the notice with the derived software. I only wish to know if any other solution is derived from this extension, nothing else. MIT License is very popular on GitHub [68] . To my knowledge, using this license is in no conflict with any other license, including the license of PostgreSQL DMBS itself. There is no obligation for the user to

add a notice anywhere if the extension is used in unmodified form as standalone software.

- **Specific schema.** All the schema objects that are a part of the extension should be created within one schema, which name is predefined by the extension. It ensures that the user cannot bundle the extension schema objects with the schema objects in some already existing schema (like *public* that is always automatically created). Everyone will know where to look for the schema objects.

5.4 Domain Model

The domain model diagram is presented in two parts as Figure 4 and Figure 5 for more the better visual clarity.

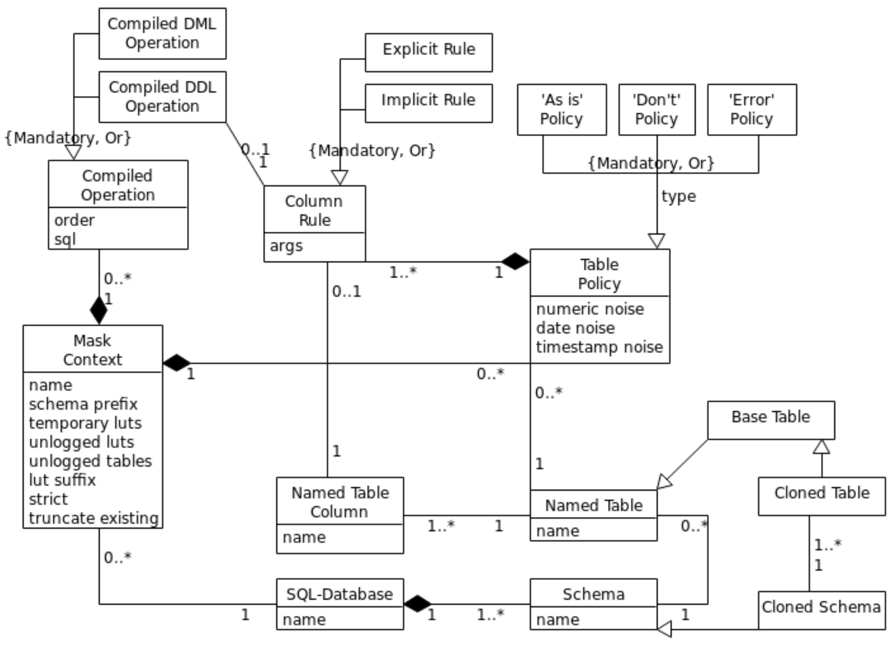


Figure 4: Extension domain model - part 1

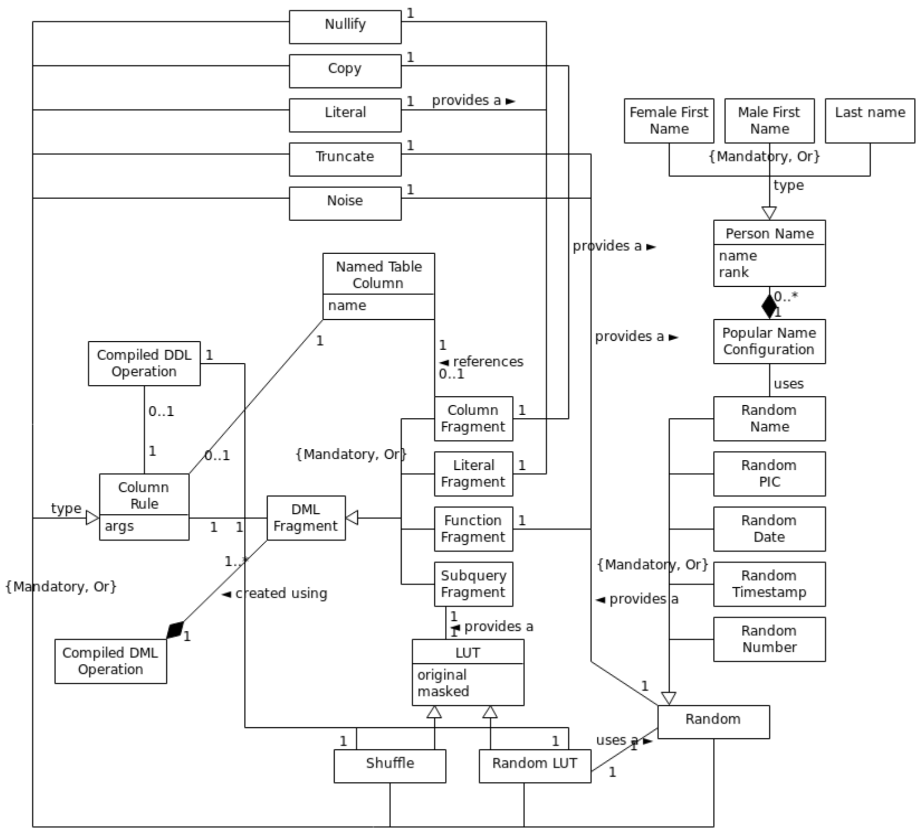


Figure 5: Extension domain model - part 2

Table 4: Description of the classes of domain model

Object Name	**Description**
Base Table	A base table is a named table of a SQL-database that is not defined in terms of other tables. There is the `CREATE TABLE` statement for creating such tables in SQL.
Named Table	In this context a base table that can be accessed by its name. Other types of named tables (views) are also defined in the SQL standard [13] . In the implemented extension all the masking rules must be applied to base tables, not to views or materialized views.
Named Table Column	A named component of a table with a data type, a nullability characteristic, and a possible default value.
Schema	A named collection of database schema object descriptors. It is the namespace for the schema objects. There are database objects that are schema objects (for instance, base tables) and database objects that are not (for instance, roles).

Object Name	**Description**
SQL-Database	SQL-Database is an organized collection of data, structures for holding the data, and ecosystem of supporting database objects that can be accessed, modified, and managed using SQL.
Mask Context	Context, in which masking settings for tables and columns are valid. A database can have multiple mask contexts.
Table policy	Table policy defines what masking operations are done by default and sets default values for various operations. Table masking policy is specific to a schema and a named table.
‘As is’ policy	A type of table masking policy that sets an implicit column rule for all columns not defined by the user to ‘copy’.
‘Don’t’ Policy	A type of table masking policy that sets an implicit column rule for all columns not defined by the user to ‘nullify’.
‘Error’ policy	A special type of table masking policy that terminates the masking process if any implicit column rules are found.
Explicit Rule	A column rule that is defined by the user.
Implicit Rule	A column rule that is defined by the table policy.
Column Rule	Column masking rule is a rule that defines the masking technique used for the specified column of the table using a DML fragment.
DML Fragment	A component of a compiled DML operation that defines the data to be inserted to the cloned table during masking.
Literal Fragment	A literal that represents `NULL` (missing data) or a constant value.
Column Fragment	A reference to the column of the table being masked.
Function Fragment	A function call that returns a value for the masked column.
Subquery Fragment	A subquery call that returns a value for the masked column.
Compiled Operation	An operation ready to be executed during mask process. Each compiled operation is represented by a set of SQL statements. Thus “compilation” means here putting together SQL statements not producing machine code. Compiled operations are executed in specific order.
Compiled DDL Operation	A set of DDL statements such as `CREATE TABLE`.
Compiled DML operation	A set of DML statements such as `INSERT INTO`.

Object Name	Description
LUT	A lookup table that maps the unmasked (original) value to the masked value.
Shuffle	Shuffle data masking technique defined in 2.3.4.
Random LUT	A lookup table that consistently maps an unmasked value to a random value. It is a type of Substitution data masking technique defined in 2.3.3.
Random	A type of Substitution technique that generates random values for every new row, i.e. does not preserve them.
Random Name	A Random subtype that generates random person names.
Random PIC	A Random subtype that generates random Estonian personal identification codes (PIC).
Random Date	A Random subtype that generates a random date.
Random Timestamp	A Random subtype that generates a random timestamp.
Random Number	A Random subtype that generates a random numeric value.
Nullify	A type of Variable Suppression technique defined in 2.3.1 that replaces a column value with `NULL`.
Literal	A type of Variable Suppression technique that replaces a column value with a constant.
Truncate	Truncation masking technique defined in 2.3.2.
Noise	Random Noise masking technique defined in 2.3.6.
Cloned Schema	A schema that is created during masking process to contain Cloned Tables.
Cloned Table	A table that is created during masking process to contain masked data.
Person Name	Any name of a natural person. This extension is using western culture notions such as “first name” and “last name”. Thus, it may be not applicable for all contexts.
Female First Name	A first name of a female person.
Male First Name	A first name of a male person.
Last Name	A last name of a person regardless of gender.
Popular Name Configuration	An aggregated list of all person names that is used for Random Name masking technique.

Attribute definitions for the domain model are in Table 5.

Table 5: Domain model attribute definitions

Class Name	**Attribute Name**	**Description**
Mask Context	name	Masking context name. Must be unique.
Mask Context	schema prefix	Prefix for cloned schema names. Defines the name of the masked schema during cloning. Thus, for instance, if the schema of original tables is *public* and prefix is *cl_*, then the name of the schema with cloned tables will be *cl_public*. Default value is 'fake_'.
Mask Context	temporary luts	Whether lookup tables are removed after the masking is done (`TRUE`) or not (`FALSE`). Can be used for debugging purposes. Default value is `TRUE`.
Mask Context	unlogged luts	Whether lookup tables are persisted as unlogged (`TRUE`) or not (`FALSE`). Can be set only for non-temporary lookup tables. Can be used for debugging purposes. Unlogged means in this case the `UNLOGGED` attribute when the lookup table is created using `CREATE TABLE` statement. Default value is `FALSE`.
Mask Context	unlogged tables	Whether masked tables are unlogged (`TRUE`) or not (`FALSE`). Can be used for debugging purposes. Meaning of "unlogged" is described in the definition of the attribute *unlogged luts*. Default value is `FALSE`.
Mask Context	lut suffix	Lookup table (LUT) suffix. Helps to define the name of a lookup table when it is created. For example, for a column *col1* in a table named *table1* in schema *schema1*, the lookup table will be named *schema1_table1_col1_lut*, if the suffix is *_lut*. Default value is '_lut'.
Mask Context	strict	Whether to enforce equality of rules at the primary key and foreign key columns (`TRUE`) or not (`FALSE`). If the masking operation differs for any such pair of columns and the user has selected strict mode, then the process must fail. Default value is `TRUE`.
Mask Context	truncate existing	Whether to truncate data in case a table with the same name as the unmasked table exists in a cloned schema (`TRUE`) or not (`FALSE`). The process should fail if such table exists and truncation is not enabled.
Table Policy	numeric noise	The default fraction value for random noise operations

Class Name	Attribute Name	Description
		on all numeric types. Default value is 0.1.
Table Policy	date noise	The default number of days for random noise operations on date types. Default value is 5.
Table Policy	timestamp noise	The default number of seconds for random noise operations on timestamp types. Default value is 3 600.
Table Policy	type	The type of the Table Policy.
Column Rule	args	Arguments for the Column Rule.
LUT	original	The original unmasked value. For example, real person names, such as John Smith, Jane Doe, etc.
LUT	masked	A value that is used to replace the original value in all cases. If the original column contains real names, then this must contain the generated names.
Person Name	name	A part of a full name of a natural person.
Person Name	rank	Numeric value that shows the popularity rank of a name. In the context of this extension a true rank is not strictly necessary, but the ranks must start from one (1), be unique for a type, and have no blanks between values.
Person Name	type	The type of the name. Examples are 'f' for female first name, 'm' for male first name, 'l' for last name of any gender.
Schema	name	The name of the Schema.
Named Table	name	The name of the Named Table. Currently the extension supports only base tables. However, masking the content of materialized views (a kind of named derived table) would also be an option.
Named Table Column	name	The name of the Named Table Column.
SQL-Database	name	The name of the SQL-Database.

5.5 Extension Configuration Tables

The diagram that describes the configuration base tables that the extension creates automatically in the target database can be seen in Figure 6.

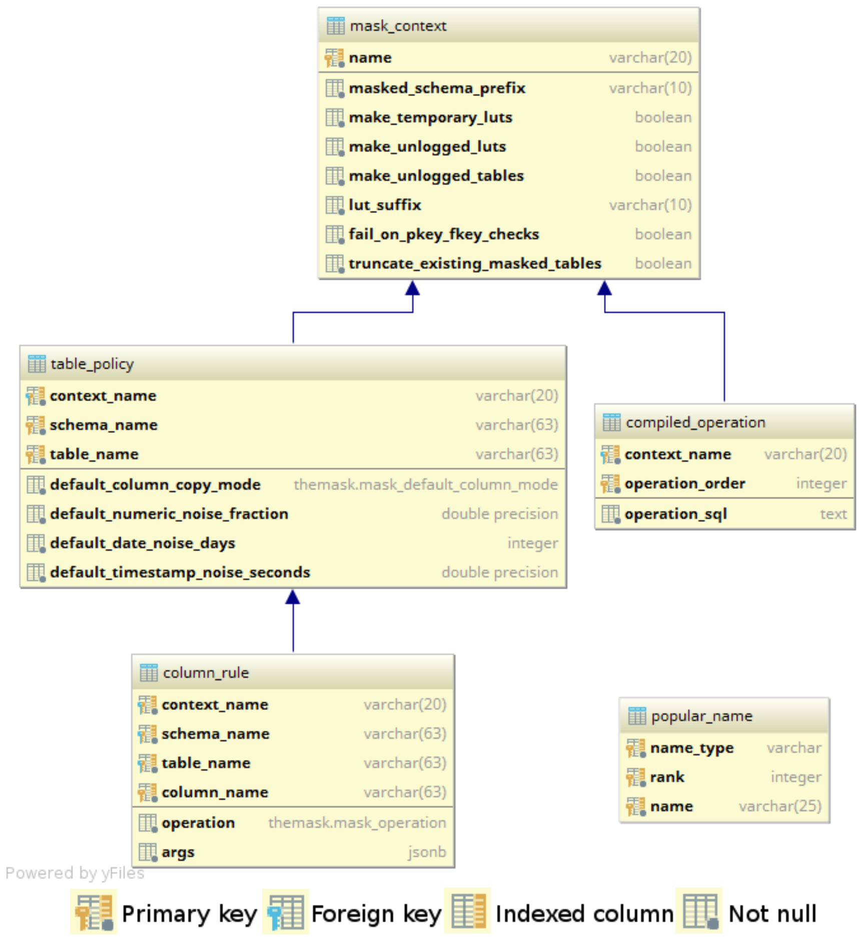


Figure 6: Extension configuration tables

Mapping between configuration tables and classes of the domain model are described in Table 6.

Table 6: Configuration table definitions

Table Name	**Domain Model Class Name**
mask_context	Mask Context
table_policy	Table Policy
compiled_operation	Compiled Operation
column_rule	Column Rule
popular_name	Person Name

Mapping between column definitions of configuration tables and attributes in the domain model is described in Table 7.

Table 7: Configuration table column definitions

Table Name	Column Name	Domain Model Class Name	Domain Model Attribute Name
mask_context	name	Mask Context	name
mask_context	masked_schema_prefix	Mask Context	schema prefix
mask_context	make_temporary_luts	Mask Context	temporary luts
mask_context	make_unlogged_luts	Mask Context	unlogged luts
mask_context	make_unlogged_tables	Mask Context	unlogged tables
mask_context	lut_suffix	Mask Context	lut suffix
mask_context	fail_on_pkey_fkey_checks	Mask Context	strict
mask_context	truncate_existing_masked_tables	Mask Context	truncate existing
table_policy	context_name	Mask Context	name
table_policy	schema_name	Schema	name
table_policy	table_name	Named Table	name
table_policy	default_column_copy_mode	Table Policy	type
table_policy	default_numeric_noise_fraction	Table Policy	numeric noise
table_policy	default_date_noise_days	Table Policy	date noise
table_policy	default_timestamp_noise_seconds	Table Policy	timestamp noise
column_rule	context_name	Mask Context	name
column_rule	schema_name	Schema	name
column_rule	table_name	Named Table	name
column_rule	column_name	Named Table Column	name
column_rule	operation	Column Rule	type
column_rule	args	Column Rule	args
compiled_operation	context_name	Mask context	name

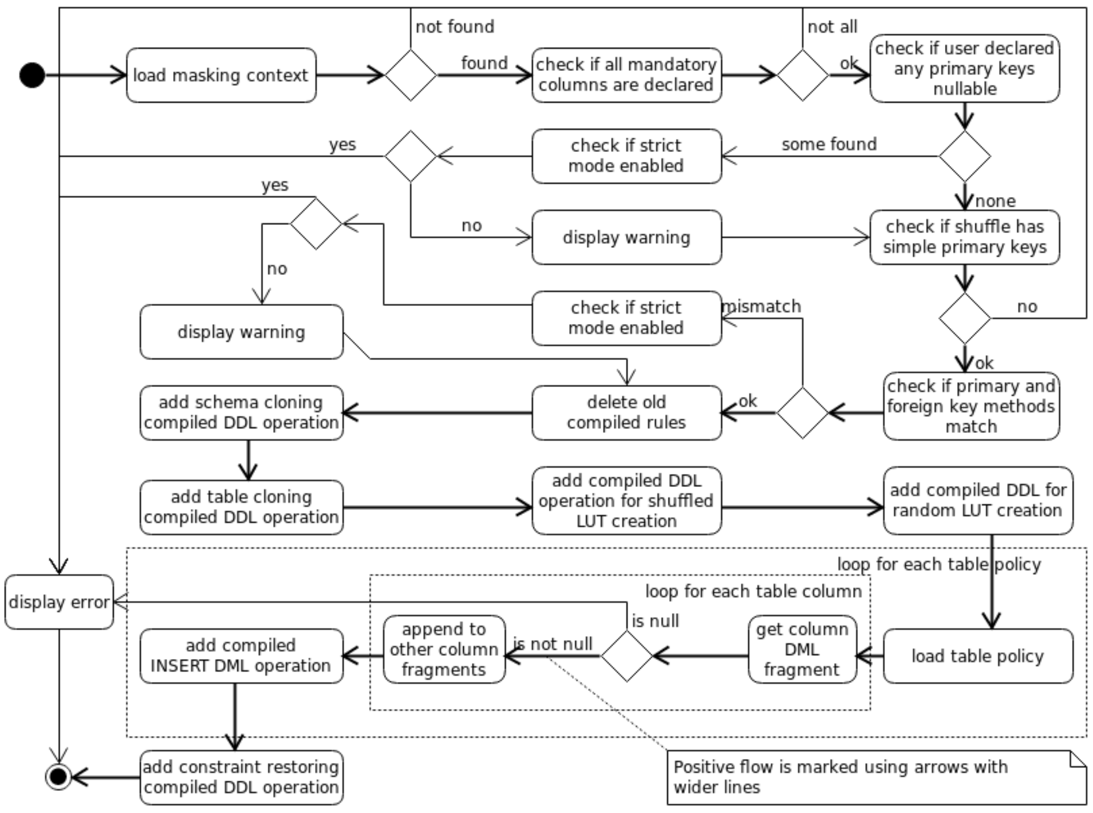
Table Name	**Column Name**	**Domain Model Class Name**	**Domain Model Attribute Name**
compiled_operation	operation_order	Compiled operation	order
compiled_operation	operation_sql	Compiled operation	sql
popular_name	name_type	Person Name	type
popular_name	rank	Person Name	rank
popular_name	name	Person Name	name

5.6 The Main Processes

Most use cases listed in 5.2 deal with the extension configuration and are of little importance to the overall masking process. There are however two use cases that are part of the core masking process – ***Rule Compilation*** and ***Masking Execution***, defined in 5.2.11 and 5.2.12, respectively. A more detailed description for each one will follow.

5.6.1 Rule Compilation

Rule compilation is essential to the process of data masking implemented in this extension. Rule compilation means automatic creation of a SQL script (a set of SQL DDL and DML statements), the execution of which creates tables with masked data. The script generation process takes into account data that is registered in the configuration tables. Understanding how it works will help understanding the overall method of operation of the extension. Figure 7 visualizes the process.



load masking context
not found
found
check if all mandatory columns are declared
not all
ok
check if user declared any primary keys nullable
some found
none
check if strict mode enabled
yes
no
display warning
check if shuffle has simple primary keys
no
ok
check if primary and foreign key methods match
mismatch
ok
check if strict mode enabled
yes
no
display warning
delete old compiled rules
add schema cloning compiled DDL operation
add table cloning compiled DDL operation
add compiled DDL operation for shuffled LUT creation
add compiled DDL for random LUT creation
loop for each table policy
loop for each table column
load table policy
get column DML fragment
is null
is not null
append to other column fragments
add compiled INSERT DML operation
display error
add constraint restoring compiled DDL operation
Positive flow is marked using arrows with wider lines

Figure 7: Rule compilation process

Rule compilation process starts with the user of the extension invoking function `themask.compile_rules('`*`masking_context_name`*`')` and ends with either an error or with all operations compiled and added to the list of compiled operations. Errors and warnings are printed using `RAISE EXCEPTION` and `RAISE WARNING` statements, respectively. The descriptions of the process steps follow.

1. It is checked if the given context exists. If not, then the process terminates.

2. It is checked if any table exists with an Error table policy and a column count that differs from the number of explicitly configured column rules. If they do exist, then an error is given and the process terminates.

3. It is checked if some table has a primary key column that has Nullify operation defined on it. Primary key values cannot be missing according to the rules of SQL. Thus, if at least one table exists and the user has chosen strict primary key

validation policy, then the operation terminates with an error. Otherwise a warning is displayed.

4. It is checked if any table contains a shuffled column and its primary key is composed of multiple columns. This is not currently supported. An error is printed and the process terminates.
5. It is checked if any table contains a column that is part of a foreign key and the masking method differs from the method of the corresponding primary key column. If at least one such table is found and the user has chosen strict primary key validation policy, then an error is presented and the process terminates. Otherwise a warning is displayed. It is expected that foreign keys only reference primary keys. Rules of SQL allow also referencing `UNIQUE` constraints.
6. All previous compiled operations are deleted.
7. Compiled DDL operations for schema and table cloning are added to the compiled operations table.
8. Compiled operations for shuffled LUT generation and population are added to the compiled operations table.
9. Compiled operations for random LUT generation and population are added to the compiled operations table.
10. For each table an `INSERT` statement is generated, in which the values to be inserted are, depending on the column masking technique, either:
    - reference to the corresponding column of the unmasked table (*copy* masking operation),
    - `NULL` (*nullify* operation),
    - a literal (*literal* operation),
    - a function call (*noise*, *random*, *mask*, and *truncate*),
    - a subquery (*random_lut*, *lut* and *shuffle*).

If the user has not specified enough parameters (see Appendix 1 – Configuration of Column Rules for more details), then the process aborts and the user is presented with an error message.

11. Finally, a compiled DDL operation is added to restore the constraints of all tables (currently a stub that does nothing, see 5.10).

For more clarity I have added a few code samples to demonstrate what compiled DDL statements appear for a test table.

Consider a table, defined by DDL statement in Figure 1.

```
CREATE TABLE public.test_table (
  id BIGSERIAL PRIMARY KEY,
  customer_name VARCHAR(50) NOT NULL,
  purchase_date DATE NOT NULL
);
```

Figure 1: Test table DDL

A test masking context and column rules are added using the statements listed in Figure 2. The rules for the columns are *copy* (`id`), *random_lut* of type *name* that generates random full names (`customer_name`), and noise (`purchase_date`).

```
SELECT themask.add_context(context_name := 'test_context');
SELECT themask.add_column_rule(_context_name := 'test_context',
                               _schema_name := 'public',
                               _table_name := 'test_table',
                               _column_name := 'id',
                               _operation := 'copy');
SELECT themask.add_column_rule(_context_name := 'test_context',
                               _schema_name := 'public',
                               _table_name := 'test_table',
                               _column_name := 'customer_name',
                               _operation := 'random_lut',
                               _args := '{"type": "name"}');
SELECT themask.add_column_rule(_context_name := 'test_context',
                               _schema_name := 'public',
                               _table_name := 'test_table',
                               _column_name := 'purchase_date',
                               _operation := 'noise');
```

Figure 2: Test table masking rules

Invoking the compilation by calling the function `themask.compile_rules(_context_name := 'test_context')` produces compiled operation records listed in Figure 3 .

```
SELECT themask._create_masked_schemas('test_context')
SELECT themask._create_masked_tables('test_context')
SELECT themask._create_lut('test_context', 'public', 'test_table',
   'customer_name')
SELECT themask._populate_lut('test_context', 'public', 'test_table',
   'customer_name', 'themask.random_full_name()')
INSERT INTO fake_public.test_table (id, customer_name, purchase_date)
   SELECT a.id AS id,
     (SELECT lut.masked FROM public_test_table_customer_name_lut lut
   WHERE lut.original = a.customer_name) AS customer_name,
   themask.random_noise(a.purchase_date, 5::INT) AS purchase_date
   FROM public.test_table a
SELECT themask._restore_constraints('test_context')
```

Figure 3: Test table compiled operations

5.6.2 Masking Execution

Masking Execution process is a relatively straightforward and is described in Figure 8.

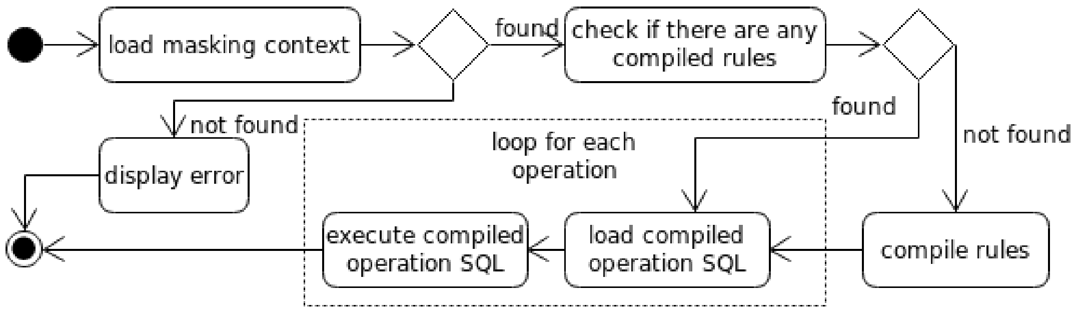


Figure 8: Masking execution process

Rule masking execution process starts with the user of the extension invoking function `themask.run('`*`masking_context_name`*`')` and ends with either an error or with all masking operation successfully executed. Errors specific to this process (meaning not system errors) are printed using `RAISE EXCEPTION`. The descriptions of the process steps follow.

1. It is checked if the given context exists. If not, the process terminates.

2. It is checked if there are compiled operations present for the given masking context. If not, then the compilation process is executed (see section 5.6.2).

3. SQL statement for each compiled operation is loaded and executed. If the statement execution fails, then a system error is shown and the process terminates. Errors in executing SQL statements are not logged by this process.

5.7 On Similarity To Commercial Data Masking Solutions

Of all the solutions mentioned in the third chapter, the created extension can be compared to Data Masking and Subsetting Pack by Oracle [46] in terms of the overall method of operation:

- It is a part of the DBMS (i.e. built-in).
- It is a static data masking solution.
- The user clones the production database or restores one from a backup.
- After configuring the masking definition a script is produced.
- The script is run on the unmasked data.
- Masked data is transferred manually to non-production environments as a database dump file.

Naturally, the solution offered by Oracle has the following advantages compared to this extension:

- A graphical user interface.
- An extensive masking format library.
- Data transfer automation via so called *pump* process [46] .

But the overall principle is similar.

5.8 Highlights on the Implementation Details

Due to non-functional requirements, the implementation is limited to using SQL and PL/pgSQL. PL/pgSQL does not require a separate binary for each platform and is installed on every distribution by default. There are a few things to consider when implementing logic in these languages.

The reader is encouraged to inspect the source code of the extension on its project page on Gitlab: https://gitlab.com/thodt-md/themask (working branch is devel as of May 2017).

5.8.1 Coding Best Practices

While PostgreSQL is not an object-oriented DBMS, it does support a form of method overloading [69] . Different methods may have the same name as long as the input parameters have different types or there is different number of parameters. This is a useful tool to avoid unnecessary code fragmentation.

PostgreSQL does not allow us to create packages to put together related functions like Oracle Database [70] does. One can use schemas to organize schema objects, but internal objects such as functions cannot be hidden from the user, except using security constraints. Internal means here internal to the implementation, i.e. functions that other functions use internally and users should not invoke these directly. Since data masking is a task for DBAs, who have superuser privileges, this form of implementation detail hiding cannot be used and one must consider using a form of convention.

I chose Python-like naming convention [71] and decided to prepend underscores (character "_") to the names of schema objects that are not to be used directly by the user of the extension. Underscores are also prepended to the names of function parameters and declared variables to avoid naming clashes with object identifiers like table and column names.

All database objects of the extension are in schema named `themask`. The schema name is hard-coded and cannot be changed by the user without code modification.

I have used the following metadata attributes for the written functions (more details on each of them are in the official manual [69]).

- `IMMUTABLE` – if a function does no database lookups, uses only information already present in its arguments, and returns the same results for given arguments. In case of repeated calls of such function with the same arguments the DBMS can answer to the request more quickly because it can use the memorized function value. In total 15 functions were marked `IMMUTABLE`.

- `STABLE` – if a function does not modify the database and within a single scan returns the same result, but its result may change across SQL statements. Functions that have read-only logic based on `SELECT` statements can be in most cases marked `STABLE`. 21 functions functions were marked `STABLE`.

- `VOLATILE` – if the function value can change even within a single table scan or if the function logic relies only on its side-effects. All functions dealing with random value generation, DML, and DDL statements were marked `VOLATILE`, totaling 53.

- `STRICT` or `RETURNS NULL ON NULL INPUT` – the function returns `NULL` if any of its arguments is `NULL`. With this modifier any such invocation will be replaced with `NULL` directly. By default a function is always called and calculated regardless of arguments. 55 functions were marked `STRICT`.

There are also the following function attributes worth mentioning from the manual for the `CREATE FUNCTION` statement [69] , which were not used by me, i.e. the DBMS uses default values in case of these.

- `ROWS` *`N`* – the number of rows returned from a set function. Although there is one function that returns a set (an internal function `themask._get_mandatory_column_counts()`), the default value of 1000 was deemed suitable.

- `LEAKPROOF` – indicates that a function has no side-effects and reveals no information about its arguments other than by returning a value. The intended purpose of this attribute seems to be safeguarding against data leaks. Functions marked as `LEAKPROOF` can be executed on rows before applying security constraints. Careful evaluation must be made before a function can be marked `LEAKPROOF`. By default the function is not `LEAKPROOF`.

Choosing the correct attributes allow the query planner to memoize the result when possible [69] .

Another performance related remark is that invoking PL/pgSQL from SQL context incurs a context switch penalty [72] . Thus, it is best to avoid calling PL/pgSQL functions in complex queries. The extension relies on using SQL-only functions and subqueries for data masking. These are used in the `INSERT` statements that add data to the tables that must contain masked data..

The extension will be installed and used by database administrators who have superuser privileges. Thus, it is suitable to use the default `SECURITY INVOKER` setting of functions. In this case the user of the function must have privileges to execute the functions as well as to perform all the activities that the function conducts.

5.8.2 Code Structure

Like it was mentioned in the section 4.1, PostgreSQL expects the code of the extension (extension script) to be in a single file. The implemented extension contains:

- 87 functions,
  - 42 functions in SQL language,
  - 45 functions in PL/pgSQL language,
- 7 views,
- 5 (base) tables,
- 2 enumeration types.

This amounts to over 2100 lines of code (counted from the resulting SQL script that is produced by GNU make, more is explained in section 5.8.3). Storing this in one file would be a major maintenance problem. For example, if many developers modify a single file, then they shall have to merge their changes in the source control system regardless whether the changes were in related parts of the file or not.

The code was written in IntelliJ IDEA 2017.2 EAP development environment. Two spaces were used as the indentation character. Each line had a limit of 120 characters maximum. Automatic code formatting was used.

It was decided to store views, tables and enumeration types in separate files and to combine functions into files by name (regardless to whether they are "public" or "private" by convention). Compared to 1:1 approach, when a single file would hold the `CREATE` statement of a single database object, this yielded overall reduction of SQL script files to 75 (*vs.* 87 + 7 + 5 + 2 = 101) with the largest file having less than 200 lines of code.

Extension source code files were placed a directory structure described in Table 8.

Table 8: Directory tree for extension SQL script files

Directory	**Description**
`├── routines`	Root for all functions. No functions here *per se*.
`│   ├── internal`	Root for internal extension functions.
`│   │   ├── checks`	Functions that perform runtime checks.
`│   │   ├── compilation`	Functions that generate compiled operation records.
`│   │   ├── process`	Functions that organize masking (do table DDLs, etc).
`│   │   └── util`	Utility functions.
`│   ├── management`	Management functions (define rules, compile, run).
`│   └── public`	Utility functions that can be used separately from masking.
`├── tables`	Extension configuration tables.
`├── types`	Extension types (enums).
`└── views`	Extension views.

5.8.3 Packaging and Distribution

While it is necessary to divide the code of the extension into smaller parts for maintainability, the requirement of a single file is still present. Thus, it must be possible to somehow concatenate files in the right order to produce a single artifact.

PostgreSQL vendors suggest using a subsystem called PGXS, which offers portable build infrastructure based on GNU Makefiles [73] . Makefiles are language-agnostic and in fact may be used for other purposes than building artifacts. For example, these can be used for documentation generation, automated testing, software installation, and software removal [74] .

Makefiles can have variables, rules and targets. A *variable* is a name defined in the script that represents a string value. A *rule* tells make what commands to execute in order to build a *target*. It also specifies the dependencies of the target – other targets or source files that the rule uses for input.

Any line that starts with the character "#" is a comment.

A few excerpts of the makefile used for the extension are given in Figure 4.

```
EXTENSION = themask
EXTVERSION = $(shell grep default_version $(EXTENSION).control | \
               sed -e "s/default_version[[:space:]]*=[[:space:]]*'\
([^']*\)'/\1/")
...
typesdir = sql/types/
...
typesdep =  ${typesdir}mask_operation.sql \
            ${typesdir}mask_default_column_mode.sql
...
all: sql/$(EXTENSION)--$(EXTVERSION).sql

sql/$(EXTENSION)--$(EXTVERSION).sql: ... ${typesdep} ...
      cat > $@ $^

DATA = sql/$(EXTENSION)--$(EXTVERSION).sql
EXTRA_CLEAN = sql/$(EXTENSION)--$(EXTVERSION).sql
...
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Figure 4. Makefile example

Here we can see variables `EXTENSION` and `EXTVERSION`. The former is required by PGXS. The latter is added so that the current version of the extension and the correct extension script file name is derived from the `default_version` variable of the control file. The `DATA` variable contains the names of the targets that are copied together with the control file to `SHAREDIR/extension` directory. In case of this extension, this is only `themask--0.0.1.sql`. `EXTRA_CLEAN` is to mark the latter file as a subject for removal when `make clean` is invoked and the build process is to be restarted.

There are 75 SQL files for creating database objects that are part of the extension and two more files of extension infrastructure-specific code (like marking configuration tables as a part of the database backup for `pg_dump`). These files must be concatenated in a specific order to produce one valid script. Writing them together with relative paths as the `sql/$(EXTENSION)--$(EXTVERSION).sql` target dependencies would be too verbose. That is why they were split into variables grouped by subdirectories listed in Table 8 to avoid code duplication and possible errors if a file is to be renamed or moved. For example, enumeration types directory is referred to as `typesdir`. All files in it are then listed in the correct order and assigned to the variable `typesdep`. It is then added to the main SQL target as a dependency. Slashes ("\") mark that a variable spans multiple lines.

The command `cat > $@ $^` concatenates all dependencies to a single script.

`include $(PGXS)` statement extends the makefile with PGXS-specific build targets and other infrastructure needed to install the extension.

Unfortunately, using PGXS directly means that the database server must have both PGXS and GNU make installed for the installation to succeed. I will likely switch to other means of packaging and distribution in the future, like publishing to the PostgreSQL Extension Network [43] . The reason why this extension is not published to PostgreSQL Extension Network yet is because the extension is in proof-of-concept stage with some issues (see part 5.10 for more details) to be resolved before it can be used as a standalone solution.

5.9 Installation

Installing the extension involves the following steps.

1. Checkout the extension code to a directory from `devel` branch.
2. Invoke GNU make in the directory with the source code to build and install the extension: `make && make install`.
3. In the target database the user must invoke the statement `CREATE EXTENSION themask`.
4. Random name generator needs a sufficiently large list of male and female first names, as well as last names. For this purpose the list of popular names was taken from open data released by the United States Census Bureau and bundled with the extension source code as CSV files [75] .
   To install them a loading script must be executed using the `psql` command-line tool: `psql -vworkdir="`*`/path/to/data/names`*`" -U postgres -d `*`database_name`*` -a -f `*`/path/to/data/names/`*`load_census_names.sql`

This step is optional if other source of random names is used, or if random names are not to be generated.

5.10 Issues and Limitations

The first publicly available version of the extension has limitations, which the users must know.

The user may report any other issues and defects not listed here at the extension Gitlab page: https://gitlab.com/thodt-md/themask/issues.

5.10.1 Not Directly Usable by Client Applications

The extension works by cloning schemas for the masked tables in the same database as the unmasked table schemas. Cloned schemas have different names than the schemas of the unmasked tables. This means that any queries referencing the unmasked schemas will not work on masked data directly.

I have not figured out an elegant way to overcome this problem, other than to rename the schemas after the masked data dump is transferred from the production database to other environments. This is a relatively simple task that can be done by the users of the masked data by issuing statements like `ALTER SCHEMA fake_schema1 RENAME TO schema1`.

In the future I plan to do research on integrating the solution with other extensions like pglogical [76] , dblink [77] or postgres_fdw [78] that provide a way to transfer data to external databases. This may possibly solve other issues mentioned in this part.

5.10.2 Incomplete Transfer of Table Details

Only column types and comments are transferred to the cloned tables. This means that any constraints (`PRIMARY KEY, UNIQUE, FOREIGN KEY`, `NOT NULL`, and `CHECK`), default values, triggers, rules, indices, and grants are not transferred.

Idea for the current implementation was to apply constraints after all the data is masked. The primary reason for this is that some masking operations, say Variable Suppression and Truncation, may render constraints unenforceable. However, this may not be an obstacle for the use of the extension and masked data.

Foreign key constraints make rows of a table depend on the rows or from the rows of the same table in case of recursive relationships. This leads to the problem of ordering:

- if the masked tables keep foreign key constraints, then the data must be inserted in the correct order. This will fail if a primary key was suppressed or rendered otherwise non-unique;
- if the constraints are applied after the data has been masked, yet again it is to be done in the correct order. The benefit of this option is that the data is still copied as masked and thus can be used;

One possible solution for applying foreign key constraints would be to construct a directed graph of tables and start applying primary key constraints from nodes that have no dependencies, then the corresponding foreign keys, etc.

Another possible solution is to let the user decide the order, in which the constraints are applied.

Cases when references are circular are yet another area of concern.

Future versions will most likely transfer the simplest metadata that does not reference anything beyond table columns (`PRIMARY KEY`, `UNIQUE`, `NOT NULL`, `CHECK` constraints, secondary indexes, default values and security constraints).

5.10.3 No Object Types Besides Base Tables

Base tables are named tables that are not defined in terms of other tables. Base tables are in the literature often simply called tables. By design only table and table columns are transferred to masked schemas. Transferring other database objects may require introspecting their definition to replace references to the unmasked tables.

The current implementation also does not allow us transfer materialized views that are a kind of derived tables.

5.10.4 Incomplete Support for Composite Primary Keys

Shuffling operation does not currently support tables with composite primary keys. An erroneous decision was made to reuse LUT DDL code generation (from `themask._create_lut()` function), which implies a single column for a primary key.

5.10.5 Inefficient Shuffling

Shuffling LUT population is done in three stages:

1. Primary key values of the table are inserted to a temporary table using a surrogate `BIGSERIAL` key.
2. Shuffled column value is inserted to another temporary table using `ORDER BY random()`, which is considered inefficient by itself [79] .
3. Contents of both tables are inserted to the shuffled LUT joined by the surrogate key values.

This implementation may be naive, yet is working. Further research is needed to implement the shuffling more efficiently. One area of research would be windowing functions and common table expressions.

5.10.6 No Option to Link the Techniques

The current implementation does not give an obvious way to link or nest masking methods of different columns. One cannot, for example, add a random date and use it to compute a random personal identification code. Each column can have only a single rule.

5.10.7 Security and Roles

This extension is designed to be used by DBAs with superuser privileges. This means that any user without a superuser role cannot invoke the masking process, view or alter its configuration. This is unlikely to change in the future.

However, as a side effect of issue 5.10.2, the cloned schemas and tables cannot be accessed by an ordinary user (like the user a web application typically uses to access the database). This issue may be solved in the future together with the issue 5.10.2, or a script-like solution can be implemented like the one done in [60] .

The user may report any other issues and defects not listed here at the extension Gitlab page: https://gitlab.com/thodt-md/themask/issues.

5.10.8 Identifier Name Length

The current implementation expects the maximum identifier length to be 63 characters (default value for most PostgreSQL distributions). This is determined by the value of `NAMEDATALEN` parameter, as described in the official manual, and can be modified before the PostgreSQL server binary is built [80] .

5.10.9 UNIQUE Referencing

The current implementation does not support foreign keys referencing `UNIQUE` constraints, only `PRIMARY KEY` constraints.

5.10.10 Schema Changes Not Detected

If the masked schema changes (a masked table/column is added, renamed, or removed), then the user must recompile the rules. I am not aware of any means to track all schema changes (especially database table/column removal). The user is responsible for the rules to be up to date with the schema. ‘Error’ table masking policy forces the user to explicitly define all columns for a masked table, which may help isolating cases when a schema change has occurred.

6 Performance Evaluation

This part describes the tests done on a a small data set to gauge the performance of the implemented extension. The data was taken from Stack Exchange Data Dump [81] (more specifically, the dump from *askubuntu.com*) and loaded to the test database using stackexchange-dump-to-postgres tool [82] . The latter had an issue I had to resolve before the loading succeeded [83] .

6.1 Test Data Schema

In order to test the performance of the created extension, three tables were chosen from the test data set: *users*, *posts* and *comments*. Their structure is described using Figure 9.

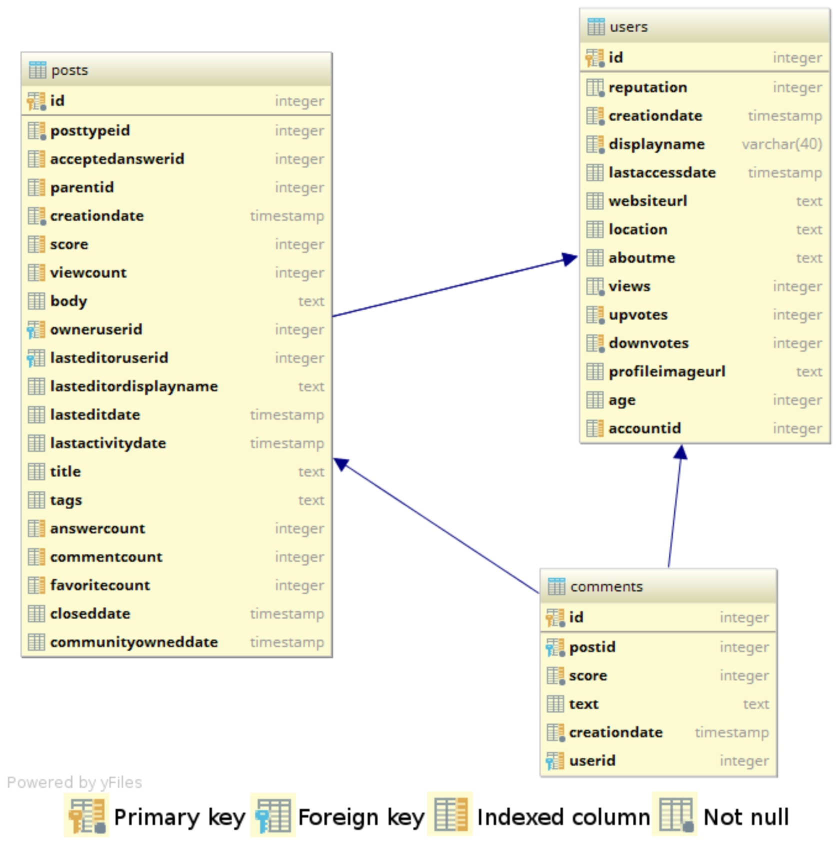


Figure 9: Test data schema

The description of the columns is not strictly relevant to this test and was omitted from the thesis for brevity. It can be found on various resources about the data dump [84] .

Although the schema does not have any foreign key constraints, I decided to add four foreign key constraints to keep testing more realistic. I have seen systems where foreign key constraints were dropped to accommodate for high loads, but most systems do still use these.

The tables, however, do have in total 22 secondary indexes. There are 11 columns with `NOT NULL` constraints. There are 865 066 rows in *comments* table, 598 530 rows in *posts* table, and 420 227 rows in *users* table.

6.2 System Setup and Methodology

Next, I explain the experiment to make it possible for interested parties to repeat it.

6.2.1 Test System Hardware and Software Setup

The test was performed on a Dell Latitude E6420 laptop with Intel Core i5-2520M CPU (2 cores, 4 threads, 3 MiB cache), 8 GiB of DDR3 SDRAM memory, and a Samsung 850 EVO 500GB solid-state drive.

PostgreSQL 9.6 DBMS was used in a Docker container running on Xubuntu 16.04 LTS operating system. `/var/lib/postgresql/data` directory of the container was mounted on a location of the root file system.

6.2.2 Test Methodology

Table 9 describes the masking techniques chosen by me to test the performance of the extension. Only masking techniques which implementations differ significantly from each other are chosen for the test. For example, Substitution (implemented as operations *random* and *random_lut*) and Noise Addition (*noise*) for different numeric types differ only by the range of the type (for temporal types it differs essentially only by the type to which the function result is cast in the `INSERT` statement). The tables and columns were chosen arbitrarily to accommodate the masking technique to be tested.

Table 9: Masking configuration for performance test

#	Table	Column	Operation	Arguments as JSON
1	users	displayname	random_lut	`{"type": "name"}`
2	users	displayname	random	`{"type": "name"}`
3	users	displayname	random	`{"type": "estonian_pic"}`
4	users	age	random	`{"range_start": 10, "range_end": 80}`
5	users	location	shuffle	`{}`
6	users	aboutme	mask	`{"left_offset": 10, "right_offset": 0}`
7	users	accountid	literal	`{"value": "0"}`
8	users	reputation	noise	`{}`
9	users	creationdate	random	`{}`
10	comments	creationdate	random	`{}`
11	comments	text	truncate	`{"length": 5, "direction": "right"}`
12	comments	text	shuffle	`{}`
13	comments	creationdate	truncate	`{"precision": "year"}`
14	posts	title	nullify	`{}`
15	posts	viewcount	noise	`{}`
16	posts	score	random	`{"range_start": 10, "range_end": 80}`
17	posts	creationdate	random	`{}`

The tests were run one operation at a time. Table policies were set to “as is”, meaning that all other columns were copied unmasked. One test without any applied rules was done to set a baseline (further referred to as Baseline test). It is the amount of time it takes to simply copy the unmasked data to the cloned schema. Since the current implementation does not transfer any table details, except column types and comments, additional code was added that restores them manually.

Time was measured starting from `themask.run()` and until all constraints were re-applied by the script. After each run the masked schema was dropped and `VACUUM FULL` statement was executed. All tests could in theory be run on a single table, but I decided to copy all three every test execution to make the test more realistic and to add more load.

Each test (including the baseline test) was repeated five times, and a mean was calculated as the result. These values are present in Table 10.

The test script template can be seen in Appendix 2 – Performance Test Script Template. Secondary index DDL statements were taken from the source code of the loading utility (*Comments_post.sql*, *Posts_post.sql*, and *Users_post.sql* files in particular) [82] .

6.3 Performance Results

The results of the tests are given in Table 10. Row #0 presents the results in case of the baseline test, which copies all the data without masking.

Table 10: Performance test results

#	**Masking Time**	**Constraint/Index Restore Time**	**Total Time**		**Baseline Adjusted Total Time**[1]	
unit	**ms**	**ms**	**ms**	**min**	**ms**	**min**
0[2]	4 553	12 921	17 474	0.29	0	0.00
1	201 080	13 045	214 125	3.57	196 651	3.28
2	183 835	12 954	196 789	3.28	179 315	2.99
3	90 620	12 869	103 489	1.72	86 015	1.43
4	4 585	12 967	17 552	0.29	78	0.00
5	8 877	12 928	21 805	0.36	4 331	0.07

[1] Difference between Total Time of the test and the Total Time of the Baseline Test.

[2] Baseline Test – test that copies all the data without masking.

#	Masking Time	Constraint/Index Restore Time	Total Time		Baseline Adjusted Total Time	
6	5 867	12 971	18 838	0.31	1 364	0.02
7	4 201	73 137	77 338	1.29	59 864	1.00
8	8 669	12 960	21 629	0.36	4 155	0.07
9	5 200	12 990	18 190	0.30	716	0.01
10	5 682	13 058	18 740	0.31	1 266	0.02
11	3 269	12 542	15 811	0.26	-1 663	-0.03
12	15 820	12 747	28 567	0.48	11 093	0.18
13	4 775	13 192	17 967	0.30	493	0.01
14	4 308	12 853	17 161	0.29	-313	-0.01
15	7 457	12 955	20 412	0.34	2 938	0.05
16	4 678	13 180	17 858	0.30	384	0.01
17	5 319	13 359	18 678	0.31	1 204	0.02

After getting the results the following conclusions were made:

- Random full name generation (and possibly random PIC generation) code must be revised. It takes slightly less than 0,5 ms to generate a random name. This is too much compared to other masking operations.
- Replacing a value with a constant (value 0) in test case #7 lead to indexing becoming noticeably slower. Since the underlying index is a hash-based index, this may be due to hash collisions. Consequently such masking operations should not be used on columns with `HASH` indexes (or the index type must be changed).
- Most masking operations gave little to no impact compared to all tables being copied unmasked. In some cases copying and at the same time masking data was actually faster than copying unmasked data (every result where the baseline adjusted total time is negative).

- Masking operations based on lookup tables (such as *random_lut* and *shuffle*) are noticeably slower than function-based operations, but the overhead is still manageable.

A separate test was made to confirm that masking multiple columns concurrently leads to no significant increase in overall processing time. After combining tests #2, #4, #5, #10, #12, #15, and #16 masking process was in fact almost 20 000 milliseconds faster than when each test was executed separately.

The time difference of `SELECT` queries between masked and unmasked data was negligible and query plans were identical for all queries I have tried. Thus, I deemed it unnecessary to publish them as a part of this thesis.

I acknowledge that the tests could in theory run faster on server-grade hardware and slower on a system with magnetic storage media for the root file system. I encourage the reader to perform the same tests on his/her system if the reported results raise any doubt (or seem otherwise somehow flawed).

7 Looking Back and Forward

In this chapter I describe miscalculations in my work process. I describe the ways to refactor the implemented solution. I also present the scope of future development .

7.1 Miscalculations of My Work Process

I acknowledge that after collecting theoretical information about data masking and information about the existing products on the market I started development with prototyping. I was trying to create the viable working product first. Setting the exact scope of the product and creating analysis models was afterthought. At one hand, prototyping gave valuable input to the analysis. On the other hand, I realize now that these processes should have been parallel and it would have saved time to me.

I acknowledge that some questionable decisions were made while implementing the extension. For example, `NOT NULL` constraints are dropped regardless of whether the user has chosen to use nulling out strategy or not. Code must be added to check as to whether this action is actually needed. Some constraints can be kept, for example, if all the columns they reference are copied unmasked.

Another flaw was trying to reuse my code written for consistent random substitution (operation type *random_lut*) for shuffling. Shuffling must be implemented independently and use system information functions (like `pg_get_constraintdef`, see [85]) to find `PRIMARY KEY` constraint definition.

7.2 Refactoring

I do give names to function parameters in order to make code better readable. However, if the extension function invokes another extension function, then I do not use parameter names in the invocation. In PostgreSQL it can be done, for instance, like that: `function_name(parameter_name := value).`

Instead, I rely to the order of parameters and specify arguments in the invocation in the corresponding order. Therefore, if the order of parameters in the invoked functions will change, then all the functions that invoke it have to be modified as well. Thus, it is better to use parameter names in the invocation.

I also do not use `LEAKPROOF` function attribute. It would be beneficial to determine how many functions can be marked as such. This would increase security and, possibly, performance.

As was shown in 6.3, one has to find out the cause and possibly fix slow random name generation.

7.3 Development Ideas for the Future

Although the implemented extension does work, considerable amount of additional development has to be done before it can be seen as a viable standalone data masking solution. I briefly describe the scope of the future work.

1. Resolve issues mentioned in part 5.10, some of the more critical being 5.10.2 (Incomplete Transfer of Table Details), 5.10.4 (Incomplete Support for Composite Primary Keys), and 5.10.7 (Security and Roles).
2. Implement Format-preserving Encryption for domains like credit card numbers, phone numbers, and the Estonian Personal Identification Code.
3. Implement simpler forms of data masking and random data generators for domains such as emails and addresses.
4. Research other packaging options than PGXS.

It should also be noted that the extension does not have any automated tests. Further research is needed to compare the available test frameworks in terms of using these to test PostgreSQL extensions. A test framework must be simple to use, yet it must allow testing all the aspects of the written code, including exception handling. It must also allow easy integration with the build framework. One good candidate is pgTAP [86] .

Various commercial solutions, for example Oracle's Data Masking and Subsetting pack, allow the user to choose the specific subset of the production data to be copied during masking [46] . Implementing such a feature for this extension would be beneficial. Currently all data from the chosen table(s) is masked, even if not needed afterwards. A possibility to achieve this is to allow data masking operations to use views as the source of unmasked data.

Finally, there is some work to be done on the usability of the extension. For example, it might be possible to add a hinting mechanism that provides the user of the extension a list of all the available masking methods for a particular column, based on the data type. It could be implemented either as a view or as a function, returning a table. Internally the view or the function should read the system catalogue of the database to find the types of columns.

I plan to continue working on the extension, and to gradually implement all of the aforementioned tasks. My preliminary estimate for the extra development and testing time is 750 – 1000 hours.

I also plan to use this extension at my main workplace, possibly to improve the code quality as part of my main work.

8 Summary

The purpose of this work was to design and implement an open-source data masking solution for PostgreSQL DBMS in the form of a proof-of-concept extension. This goal was achieved and the source code for the extension is available at:

https://gitlab.com/thodt-md/themask/tree/devel

The source code is licensed with the MIT license.

The thesis started with a theoretical background about data masking in the second chapter. An overview of commercial proprietary solutions was given in the third chapter.

The main part of the work started from the fourth chapter. In this implementation alternatives in PostgreSQL were described.

Chapter five focused on the design and implementation of the extension. First, the requirements were presented. After that a domain model was created based on the the requirements to describe the concepts of the solution and their relationships. Chapter five also described the base tables that the extension creates upon installation and the main processes that are essential to the operation of the extension. The extension was compared to the most similar commercial solution – Data Masking and Subsetting Pack by Oracle.

Next, extension packaging and installation was discussed. The chapter was concluded with known issues and limitations of the extension. Ways of working around the issues currently and possibly fixing them in the future were shown.

An evaluation of the extension in the form of a performance test was given in the sixth chapter.

In chapter seven the possible future work, as well as the needs to refactor the current solution were presented.

This work is useful to database administrators tasked with integrating data masking in systems they are responsible for, as well as to developers of other PostgreSQL extensions.

References

[1] DB-Engines Ranking [WWW] https://db-engines.com/en/ranking (01.05.2017)

[2] Advanced Encryption Standard [WWW] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard (07.05.2017)

[3] RFC4180 - Common Format and MIME Type for Comma-Separated Values (CSV) Files [WWW] https://tools.ietf.org/html/rfc4180 (01.05.2017)

[4] Database Administrator (DBA) [WWW] http://www.businessdictionary.com/definition/Database-Administrator-DBA.html (05.05.2017)

[5] DBMS - database management system [WWW] http://www.webopedia.com/TERM/D/database_management_system_DBMS.html (01.05.2017)

[6] Bellare M., Ristenpart T., Rogaway P., Stegers T. Format-Preserving Encryption. – International Workshop on Selected Areas in Cryptography, SAC 2009. – Lecture Notes in Computer Science, 5867, pp. 295-312 [Online] SpringerLink. LNCS (02.05.2017)

[7] JSON [WWW] http://www.json.org/ (01.05.2017)

[8] JSON Types [WWW] https://www.postgresql.org/docs/current/static/datatype-json.html (01.05.2017)

[9] Lookup table – Wikipedia [WWW] https://en.wikipedia.org/wiki/Lookup_table (01.05.2017)

[10] Rahvastikuregistri seadus – Riigi Teataja I, 2000, 50, 317 (in Estonian)

[11] PL/pgSQL – Wikipedia [WWW] https://en.wikipedia.org/wiki/PL/pgSQL (01.05.2017)

[12] RDBMS – relational database management system [WWW] http://www.webopedia.com/TERM/R/RDBMS.html (03.05.2017)

[13] Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework): ISO/IEC 9075-1:200x

[14] Information technology – Open Distributed Processing – Unified Modeling Language (UML): ISO/IEC 9075-1:2008

[15] Introduction to XML [WWW] https://www.w3schools.com/xml/xml_whatis.asp (03.05.2017)

[16] Syal, R. Abandoned NHS IT system has cost £10bn so far. [WWW] https://www.theguardian.com/society/2013/sep/18/nhs-records-system-10bn (02.05.2017)

[17] Eesti Päevaleht – Juhtkiri: ainult poliitilise tahtega suurt IT-projekti ei teosta [WWW] http://epl.delfi.ee/archive/print.php?id=77760682 (02.05.2017) (in Estonian)

[18] Holgeid K., Thompson M. A reflection on why large public projects fail. – The Governance of Large-Scale Projects, Nomos Verlagsgesellschaft mbH & Co. KG, June 2013, pp. 219-244.

[19] Von Alan, R., Hevner, et al. Design Science in Information Systems Research. – MIS quarterly, 2004, 28 (1), pp. 75-105.

[20] Ravikumar, G. K., Manjunath, T. N., Ravindra, S., Umesh, I. M. A Survey on Recent Trends, Process and Development in Data Masking for Testing. – International Journal of Computer Science Issues, 2011, 8(2), pp. 535-544.

[21] Ravikumar, G. K., Rabi, B. J., Manjunath, T.N. A Study on Dynamic Data Masking with its Trends and Implications. – International Journal of Computer Applications, 2012, 38 (6), pp. 19-24.

[22] Data Masking: What You Need to Know – A Net 2000 Ltd. White Paper [WWW] http://www.datamasker.com/DataMasking_WhatYouNeedToKnow.pdf (03.05.2017)

[23] Data Masking Best Practice – An Oracle White Paper [WWW] http://www.oracle.com/us/products/database/data-masking-best-practices-161213.pdf (02.05.2017)

[24] Mogull, R. – The Five Laws Of Data Masking [WWW] https://securosis.com/blog/the-five-laws-of-data-masking (02.05.2017)

[25] generatedata.com [WWW] http://generatedata.com/ (15.05.2017)

[26] Data Masking Made Simple with DataSunrise Data Masking Tool [WWW] https://www.datasunrise.com/data-masking-made-simple/ (03.05.2017)

[27] Mushkablat, V. – ELIMINATING COMPLINCE RISKS - DATA MASKING WITH AZURE [WWW] http://mask-me.net/Downloads/Data%20Masking%20Addressing%20Risks%20in%20the%20Cloud-new.pptx (07.05.2017)

[28] Ravikumar, G. K., Rabi, B. J., Hegadi, R. S., Manjunath, T. N., Archana, R. A. Experimental Study of Various Data Masking Techniques with Random Replacement using data volume. – International Journal of Computer Science Issues, 2011, 9(8), pp. 154-158.

[29] Коломыцев, М. В., Южаков, А. М. Защита персональных данных методом маскирования. – Захист інформації, 2013, 15 (4), pp. 382-387 (in Russian)

[30] Sarada G., Abitha N., Manikandan G., Sairam N. A few new approaches for data masking. – 2015 International Conference on Circuit, Power and Computing Technologies (ICCPCT), Nagercoil, India, March 19, 2015: Proceedings – IEEE (pp. 1-4) [Online] IEEE Xplore (17.04.2017)

[31] Muralidhar, K., Sarathy, R. Recent Advances in Protecting Sensitive Numerical Data through Data Masking. – 6th Annual Security Conference, Las Vegas NV, April 11-12, 2007: In Proceedings, pp. 41-2-41-11

[32] Muralidhar, K., Sarathy, R. ‘Easy to Implement’ is Putting the Cart before the Horse: Effective Techniques for Masking Numerical Data – 2007 Federal Committee On Statistical Methodology Research Conference, Arlington VA, November 2007 : Proceedings (pp. 5-7).

[33] Li, XB., Sarkar, S. Against classification attacks: A decision tree pruning approach to privacy protection in data mining. – Operations Research, December 2009, 57 (6), 1496-509.

[34] Emam, K. – Perspectives on Health Data De-identification [WWW] https://iapp.org/media/pdf/knowledge_center/Perspectives_on_Health_Data_De-Identification_final.pdf (03.05.2017)

[35] Isikuandmete kaitse seadus – Riigi Teataja, 2003, 26, 158 (in Estonian)

[36] Isikukood. Struktuur : EVS 585:2007. Tallinn : Eesti Standardikeskus, 2007. (in Estonian)

[37] Duncan, G. T., Keller-McNulty S. A., Stokes, S. L. Disclosure risk vs. data utility : The R-U confidentiality map. – Chance, 2001.

[38] Sweeney, L. k-anonymity: a model for protecting privacy. – International Journal on Uncertainty, Fuzziness and Knowledge-based Systems, 2002, 10 (5), pp. 557-570.

[39] Meyerson, A., Williams, R. On the Complexity of Optimal K-anonymity. – Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, Paris, France, 2004 : PODS '04, pp. 223-228.

[40] Emam et al., Globally Optimal k-Anonymity for De-Identification of Health Data. – Journal of the American Medical Informatics Associacion, September/October 2009, 16 (5), pp. 670-682

[41] What is Data Utility [WWW] http://www.igi-global.com/dictionary/data-utility/6838 (15.05.2017)

[42] Data Types [WWW] https://www.postgresql.org/docs/current/static/datatype.html (03.05.2017)

[43] PostgreSQL Extension Network [WWW] https://pgxn.org/ (10.05.2017)

[44] Dynamic Data Masking [WWW] https://docs.microsoft.com/en-us/sql/relational-databases/security/dynamic-data-masking (07.05.2017)

[45] IBM Knowledge Center - DB2 11 - DB2 SQL - CREATE MASK [WWW] https://www.ibm.com/support/knowledgecenter/en/SSEPEK_11.0.0/sqlref/src/tpc/db2z_sql_createmask.html (07.05.2017)

[46] Data Masking and Subsetting Guide [WWW] https://docs.oracle.com/database/121/DMKSB/data_masking.htm (07.05.2017)

[47] Introduction to Oracle Data Redaction [WWW] https://docs.oracle.com/database/121/ASOAG/redaction.htm (07.05.2017)

[48] FUJITSU Software Enterprise Postgres – White paper [WWW] https://www.fujitsu.com/global/documents/products/software/middleware/opensource/postgres/resources/wp-fep-V9-5-ww-en.pdf (07.05.2017)

[49] DataSunrise Database Security Suite 3.7.1 – User Guide [WWW] https://www.datasunrise.com/documents/DataSunrise_Database_Security_Suite_User_Guide.pdf (07.05.2017)

[50] Managing Algorithm Settings [WWW] https://docs.delphix.com/docs/delphix-masking/masking-engine-admin-guide/managing-algorithm-settings (07.05.2017)

[51] DATA MASKING TRANSFORMERS – Technical Brief [WWW] https://datamasking.com/wp-content/uploads/2015/03/CAMOUFLAGE-PRODUCT-SHEET-TRANSFORMERS1.pdf (07.05.2017)

[52] DataMasker by NetLtd2000 [WWW] http://www.datamasker.com/datamasker_kf.pdf (07.05.2017)

[53] IBM InfoSphere Optim Data Privacy [WWW] https://www.ibm.com/ms-en/marketplace/infosphere-optim-data-privacy/details#product-header-top (07.05.2017)

[54] Compare IBM data masking solutions: InfoSphere Optim and DataStage [WWW] https://www.ibm.com/developerworks/data/library/techarticle/dm-1211maskingsolution/dm-1211maskingsolution-pdf.pdf (07.05.2017)

[55] Protect Data Privacy by Persistently Masking Sensitive Information [WWW] https://www.informatica.com/content/dam/informatica-com/global/amer/us/collateral/data-sheet/persistent-data-masking_data-sheet_6990.pdf (07.05.2017)

[56] Protect Data Privacy by Dynamically Masking Sensitive Information https://www.informatica.com/content/dam/informatica-com/global/amer/us/collateral/data-sheet/dynamic-data-masking_data-sheet_1779.pdf (07.05.2017)

[57] IRI FieldShield Data Masking [WWW] http://www.iri.com/products/fieldshield (07.05.2017)

[58] PostgreSQL dump obfuscation (sensitive data masking) tool. [WWW] https://github.com/ostrovok-team/pgdump-obfuscator (07.05.2017)

[59] Магический Квадрант Гартнера. Технологии маскировки данных [WWW] https://www.dataarmor.ru/quadrant/ (07.05.2017) (in Russian)

[60] Jalakas, O. Data Masking and User Rights in Data Warehouse to Protect Data : Master's thesis. Tallinn University of Technology, Tallinn, 2016

[61] Packaging Related Objects into an Extension [WWW] https://www.postgresql.org/docs/current/static/extend-extensions.html (04.05.2017)

[62] Composite Types [WWW] https://www.postgresql.org/docs/current/static/rowtypes.html (04.05.2017)

[63] Inheritance [WWW] https://www.postgresql.org/docs/current/static/ddl-inherit.html (04.05.2017)

[64] CREATE TYPE [WWW] https://www.postgresql.org/docs/current/static/sql-createtype.html (15.05.2017)

[65] The Path of a Query [WWW] https://www.postgresql.org/docs/current/static/query-path.html (04.05.2017)

[66] Re: Performance PLV8 vs PLPGSQL [WWW] https://www.postgresql.org/message-id/CAHyXU0yj7KEW2f%2BNQSnNgRyo%3DH6GF5mRDjs%2BVWQ9jK1qqtNRHw%40mail.gmail.com (10.05.2017)

[67] Licenses by Name [WWW] https://opensource.org/licenses/alphabetical (15.05.2017)

[68] Open source license usage on GitHub.com [WWW] https://github.com/blog/1964-open-source-license-usage-on-github-com (08.05.2017)

[69] CREATE FUNCTION [WWW] https://www.postgresql.org/docs/current/static/sql-createfunction.html (05.05.2017)

[70] Porting from Oracle PL/SQL [WWW] https://www.postgresql.org/docs/current/static/plpgsql-porting.html (05.05.2017)

[71] PEP 8 -- Style Guide for Python Code [WWW] https://www.python.org/dev/peps/pep-0008/#id48 (05.05.2017)

[72] Conway, N. – Inside the PostgreSQL Query Optimizer [WWW] http://www.neilconway.org/talks/optimizer/optimizer.pdf (05.05.2017)

[73] Extension Building Infrastructure [WWW] https://www.postgresql.org/docs/current/static/extend-pgxs.html (06.05.2017)

[74] GNU make [WWW] https://www.gnu.org/software/make/ (06.06.2017)

[75] Frequently Occurring Surnames from Census 1990 – Names Files [WWW] https://www.census.gov/topics/population/genealogy/data/1990_census/1990_census_namefiles.html (05.05.2017)

[76] pglogical [WWW] https://www.2ndquadrant.com/en/resources/pglogical/ (10.05.2017)

[77] dblink [WWW] https://www.postgresql.org/docs/current/static/dblink.html (10.05.2017)

[78] postgres_fdw [WWW] https://www.postgresql.org/docs/current/static/postgres-fdw.html (10.05.2017)

[79] Re: Performance of ORDER BY RANDOM to select random rows? [WWW] https://www.postgresql.org/message-id/20130808085517.GA26014%40depesz.com (05.05.2017)

[80] Preset Options [WWW] https://www.postgresql.org/docs/current/static/runtime-config-preset.html (14.05.2017)

[81] Stack Exchange Data Dump [WWW] https://archive.org/details/stackexchange (06.06.2017)

[82] Python scripts to import StackExchange data dump into Postgres DB. [WWW] https://github.com/Networks-Learning/stackexchange-dump-to-postgres (09.05.2017)

[83] Creating index on a non-existing column fails #7 [WWW] https://github.com/Networks-Learning/stackexchange-dump-to-postgres/pull/7 (09.05.2017)

[84] Database schema documentation for the public data dump and SEDE [WWW] https://meta.stackexchange.com/questions/2677/database-schema-documentation-for-the-public-data-dump-and-sede (10.05.2017)

[85] System Information Functions [WWW] https://www.postgresql.org/docs/current/static/functions-info.html (12.05.2017)

[86] pgTAP [WWW] http://pgtap.org/ (10.05.2017)

Appendix 1 – Configuration of Column Rules

In order to successfully use the implemented extension the user has to know how to configure and manage it.

Configuring column masking rules is done by invoking functions `themask.add_column_rule('context_name', 'schema', 'table', 'column', 'operation name', 'parameter json')` for column rule adding and `themask.modify_column_rule('context_name', 'schema', 'table', 'column', 'operation name', 'parameter json')` for column rule modification. A column rule may be deleted and thus reverted to its implicit state defined by the table policy by invoking `themask.modify_column_rule('context_name', 'schema', 'table', 'column')`.

In Table 11 all implemented masking techniques, their parameters, and samples values are given.

Table 11: Implemented masking extension column rule configuration parameters

Technique (Section 2.3)	**Operation Name**	**Supported Column Types**	**Parameters as a JSON Sample Fragment**	**Description**
N/A (no masking)	copy	*any*	N/A	Copies the column value as-is.
Variable Suppression	nullify	*any*	N/A	Replaces the column value with `NULL`.
	literal	*any*	`{ "name": "value" }`	Casts `value` to the corresponding column data type. **Note:** the value is passed unescaped to allow more complex use cases.

Random Noise	noise	`NUMERIC,`
		`SMALLINT,`
		`INT,`
		`BIGINT,`
		`REAL,`
		`DOUBLE PRECISION`
			`{`
			`  "fraction": ` *`float`*[1]
			`}`
				Adds `±fraction × ` *`column value`* of random noise to the column value. **Note:** For fixed precision numeric types `(SMALLINT, INT, BIGINT)` the value is clamped to their respective boundaries, if overflows.

		`DATE`	`{`
			`  "days": ` *`int`*[1]
			`}`
				Adds `±days` of random noise to the column value.

		`TIMESTAMP`	`{`
			`  "seconds": ` *`float`*[1]
			`}`
				Adds `±seconds` of random noise to the column value.

Substitution (generate random value for each row)	random	`NUMERIC,`
		`REAL,`
		`DOUBLE PRECISION`
			`{`
			`  "range_start": ` *`float`*[1]`,`
			`  "range_end": ` *`float`*[1]
			`}`
				Chooses a random number between `range_start` and `range_end`.
				If no parameters are given, the range is assumed to be the natural range of the corresponding type.

		`SMALLINT,`
		`INT,`
		`BIGINT`
			`{`
			`  "range_start": ` *`int`*[1]`,`
			`  "range_end": ` *`int`*[1]
			`}`	

		`DATE,`
		`TIMESTAMP`
			`{`
			`  "range_start": "`*`date`*`",`
			`  "range_end": "`*`date`*`"`
			`}`
				Chooses a random date or timestamp between `range_start` and `range_end`. Date and timestamp values are cast from

[1] JSON specification has no separate notation for floating and integer numeric types; float here means a floating point value is accepted, int means an integer is accepted.

				string values to the corresponding types, so care must be taken that they are in the correct format. If no parameters are given, the date is between epoch [1]and current time.
		`TEXT, VARCHAR`	`{ "type": "type" }`	Type can be `"name"` or `"estonian_pic"`. The first option generates a random name, the second option generates a random Personal Identification Code.
Truncation or Cropping	truncate	`NUMERIC, SMALLINT, INT, BIGINT, REAL, DOUBLE PRECISION`	`{ "digits": int` [1] `}`	Invokes built-in `trunc(column, digits)`. This truncates numeric value precision to the passed number of digits. More information can be found in the official manual for `trunc`.
		`TEXT, VARCHAR`	`{ "length": int` [1]`, "direction":"direction" }`	Removes `length` first characters from `direction` (`"left"`, `"right"`) side of the column value.
		`DATE, TIMESTAMP`	`{ "precision":"precision" }`	Invokes built-in `date_trunc(column, precision)`.

[1] The 1-st of January, 1970 at 00:00:00 Coordinated Universal Time (UTC).

				Precision can be, for example, `"year"`, `"month"`, or `"day"`. More information on different precision values can be found in the official manual for `date_trunc`.

Masking Out	mask	`TEXT, VARCHAR`	`{`
			`"left_offset": int`[1]`,`
			`"right_offset": int`[1]`,`
			`"mask_character": "X"`
			`}`
				Replaces all characters starting from `left_offset` and ending with `length - right_offset` with `mask_character`.
				If no mask_character value is given, then character "X" is assumed.

Substitution (reuse values from another column)	lut	*Any*	`{`
			`"schema": "schema",`
			`"table": "table",`
			`"column": "column"`
			`}`
			`or`
			`{`
			`"name": "lut_name"`
			`}`
				Uses a pre-created LUT (either manually or using *random_lut* method) for mapping values from the non-masked column.
				Note: a `UNIQUE` index is expected for non-masked values in the LUT, if created manually.

Substitution (generate random values for	random_lut	All supported by **random** method.	Same as in **random** metod description.	Creates a LUT, where each value from the non-masked column is

each unique value)				mapped uniquely to a random value.
Shuffling	shuffle	*Any*[1]	`{}` or `{ "other_schema":"`*schema*`", "other_table":"`*table*`", "other_column":"`*column*`", "fkey_column":"`*column*`" }`	Shuffles the values inside the column if no arguments are given. Otherwise uses values from a shuffled column of another table that this table references by *`fkey_column`*.

1 Tables with composite primary keys are currently not supported, see 5.10.4.

Appendix 2 – Performance Test Script Template

```
SELECT themask.add_context('test');
SELECT themask.add_table_policy('test', 'public', 'comments',
   'as_is');
SELECT themask.add_table_policy('test', 'public', 'posts', 'as_is');
SELECT themask.add_table_policy('test', 'public', 'users', 'as_is');
-- operation being tested goes here
SELECT themask.compile_rules('test');
DO $$
DECLARE
  run_start TIMESTAMP := clock_timestamp();
  constraint_start TIMESTAMP;
  run_end TIMESTAMP;
BEGIN
  PERFORM themask.run('test');
  constraint_start := clock_timestamp();
  SET SEARCH_PATH = "fake_public";
  ALTER TABLE comments add CONSTRAINT comments_pkey PRIMARY KEY (id);
  ALTER TABLE comments ALTER COLUMN postid SET NOT NULL ;
  ALTER TABLE comments ALTER COLUMN score SET NOT NULL ;
  ALTER TABLE comments ALTER COLUMN creationdate SET NOT NULL ;
  ALTER TABLE posts ADD CONSTRAINT posts_pkey PRIMARY KEY (id);
  ALTER TABLE posts ALTER column posttypeid SET NOT NULL ;
  ALTER TABLE posts ALTER column creationdate SET NOT NULL ;
  ALTER TABLE users ADD CONSTRAINT users_pkey PRIMARY KEY (id);
  ALTER TABLE users ALTER COLUMN reputation SET NOT NULL ;
  ALTER TABLE users ALTER COLUMN creationdate SET NOT NULL ;
  ALTER TABLE users ALTER COLUMN displayname SET NOT NULL ;
  ALTER TABLE users ALTER COLUMN views SET NOT NULL ;
  ALTER TABLE users ALTER COLUMN upvotes SET NOT NULL ;
  ALTER TABLE users ALTER COLUMN downvotes SET NOT NULL ;
  ALTER TABLE comments ADD CONSTRAINT comments_userid_fkey FOREIGN KEY
   (userid) REFERENCES users(id);
  ALTER TABLE comments ADD CONSTRAINT comments_postid_fkey FOREIGN KEY
   (postid) REFERENCES posts(id);
  ALTER TABLE posts ADD CONSTRAINT posts_owneruserid_fkey FOREIGN KEY
   (owneruserid) REFERENCES users(id);
```

Figure 5: Performance test script template - part1

```
  ALTER TABLE posts ADD CONSTRAINT posts_lasteditoruserid_fkey FOREIGN
   KEY (lasteditoruserid) REFERENCES users(id);
  CREATE INDEX cmnts_score_idx ON Comments USING BTREE (Score) WITH
   (FILLFACTOR = 100);
  CREATE INDEX cmnts_postid_idx ON Comments USING HASH (PostId) WITH
   (FILLFACTOR = 100);
  CREATE INDEX cmnts_creation_date_idx ON Comments USING BTREE
   (CreationDate) WITH (FILLFACTOR = 100);
  CREATE INDEX cmnts_userid_idx ON Comments USING BTREE (UserId) WITH
   (FILLFACTOR = 100);
  CREATE INDEX posts_post_type_id_idx ON Posts USING BTREE
   (PostTypeId) WITH (FILLFACTOR = 100);
  CREATE INDEX posts_score_idx ON Posts USING BTREE (Score) WITH
   (FILLFACTOR = 100);
  CREATE INDEX posts_creation_date_idx ON Posts USING BTREE
   (CreationDate) WITH (FILLFACTOR = 100);
  CREATE INDEX posts_owner_user_id_idx ON Posts USING HASH
   (OwnerUserId) WITH (FILLFACTOR = 100);
  CREATE INDEX posts_answer_count_idx ON Posts USING BTREE
   (AnswerCount) WITH (FILLFACTOR = 100);
  CREATE INDEX posts_comment_count_idx ON Posts USING BTREE
   (CommentCount) WITH (FILLFACTOR = 100);
  CREATE INDEX posts_favorite_count_idx ON Posts USING BTREE
   (FavoriteCount) WITH (FILLFACTOR = 100);
  CREATE INDEX posts_viewcount_idx ON Posts USING BTREE (ViewCount)
   WITH (FILLFACTOR = 100);
  CREATE INDEX posts_accepted_answer_id_idx ON Posts USING BTREE
   (AcceptedAnswerId) WITH (FILLFACTOR = 100);
  CREATE INDEX posts_parent_id_idx ON Posts USING BTREE (ParentId)
   WITH (FILLFACTOR = 100);
  CREATE INDEX user_acc_id_idx ON Users USING HASH (AccountId) WITH
   (FILLFACTOR = 100);
  CREATE INDEX user_display_idx ON Users USING HASH (DisplayName) WITH
   (FILLFACTOR = 100);
  CREATE INDEX user_up_votes_idx ON Users USING BTREE (UpVotes) WITH
   (FILLFACTOR = 100);
  CREATE INDEX user_down_votes_idx ON Users USING BTREE (DownVotes)
   WITH (FILLFACTOR = 100);
  CREATE INDEX user_created_at_idx ON Users USING BTREE (CreationDate)
   WITH (FILLFACTOR = 100);
  run_end := clock_timestamp();
  RAISE INFO 'mask: %, reindex: %', extract(EPOCH FROM
   constraint_start - run_start),
  extract(EPOCH FROM run_end - constraint_start);
END;
$$;
```

Figure 6: Performance test script template - part2