**Оглавление**

[Введение 2](#__RefHeading___Toc701_248757517)

[Описание версионных миграций 2](#__RefHeading___Toc703_248757517)

[Преимущества использования миграций 2](#__RefHeading___Toc705_248757517)

[Риски и недостатки миграций 2](#__RefHeading___Toc707_248757517)

[Постановка задачи 3](#__RefHeading___Toc709_248757517)

[Метод решения 4](#__RefHeading___Toc711_248757517)

[Генерация данных 6](#__RefHeading___Toc713_248757517)

[Порядок вставки генерируемых данных 6](#__RefHeading___Toc715_248757517)

[Циклически связанные таблицы 6](#__RefHeading___Toc717_248757517)

[Вставка данных без учета циклов 7](#__RefHeading___Toc719_248757517)

[Принцип генерации данных через определение набора требований 7](#__RefHeading___Toc721_248757517)

[Определение множества значений для генерируемых данных 8](#__RefHeading___Toc723_248757517)

[Ограничение CHECK 8](#__RefHeading___Toc725_248757517)

[Генерация данных на основе шаблонных процессоров 9](#__RefHeading___Toc727_248757517)

[Этапы генерации данных 10](#__RefHeading___Toc921_248757517)

[Система автоматического тестирования версионных миграций 11](#__RefHeading___Toc729_248757517_%D0%9A%D0%BE%D0%BF%D0%B8%D1%8F_1)

[Основные этапы работы программы 11](#__RefHeading___Toc923_248757517)

# Введение

Можно

осуществляется

без деепричастных

так же -- заменить

является

представляют собой -- аккуратно

это — как заменять

однако — слишком часто

выше ниже — убрать

колонки — столбцы — атрибуты

я, мы — лишнее. Обезличенные предложения

1. Хранилища данных являются — большинство приложений неотъемлемой частью многих клиентских приложений. Они обеспечивают структурированное хранение данных и возможность определения схем. Реляционные базы данных, такие как PostgreSQL, широко используются для этой цели. -какой цели
2. В настоящее время управление структурами баз данных является одной из важных задач при разработке программного обеспечения. Для *этого* есть/существует множество подходов, одним из которых — *в частности* являются миграции.

## Описание версионных миграций

1. Миграции *это практичный и распр* представляют собой практичный и распространенный подход к управлению структурами баз данных. Суть его заключается в создании последовательности изменений, которые необходимо внести *в структуру* базу данных для достижения требуемого состояния.
2. Однако, как и любой другой код, написанный человеком, миграции *могут* содержать ошибки. В конечном итоге, миграции — это рукописный SQL скрипт, который может привести к непредвиденным последствиям при выполнении. Поэтому перед применением миграции необходимо тщательно ее протестировать и убедиться, что все выполняется корректно.

*Домены значений тестриование корректности/некорректности. Литература - maiers*

## Преимущества использования миграций

Использование миграций имеет несколько преимуществ: - каламбур

1. Отслеживание изменений в базе данных. Миграции позволяют отслеживать все изменения, которые были внесены *в базу данных — в неё*. Это упрощает процесс разработки и сопровождения приложения.
2. Легкость масштабирования. Миграции позволяют легко масштабировать базу данных при необходимости.
3. Работа в команде. Использование миграций позволяет легко работать в команде над общим проектом. Каждый разработчик может создавать свои миграции, которые затем будут объединены в одну последовательность.

## Риски и недостатки миграций

Некоторые из наиболее распространенных проблем, связанных с версионными миграциями, включают:

1. Потеря данных. При неправильном использовании миграций*. Так же может произойти потеря данных или* из за ошибки в исходном коде миграции может произойти потеря данных.
2. Проблемы с производительностью: если миграции не оптимизированы должным образом, они могут привести к проблемам с производительностью базы данных.
3. Сложности с *откатом*: если миграции не удалось выполнить правильно, может потребоваться откат к предыдущей версии схемы базы данных. Однако, если миграции не были правильно написаны, то откат может оказаться сложным или даже невозможным.
4. Проблемы безопасности. *При некорр исп мигр могут быть* если миграции не были правильно выполнены, это может привести к проблемам с безопасностью. Например, передача прав доступа к конфиденциальным данным для выполнения миграции может привести к утечке информации.

Таким образом, использование миграций является эффективным подходом к управлению структурами баз данных. Однако необходимо помнить о возможных рисках, связанных с *ошибками* в миграциях.

# Постановка задачи

1. Для отслеживания ошибок, возникающих в процессе миграций баз данных, существуют различные подходы, например статический и динамический анализ скриптов миграций.
2. Статический анализ скриптов миграции заключается в том, что для поиска ошибок используется только SQL-код скрипта. В данном случае не требуется наличия рабочей базы данных, на которой будут применяться скрипты. Однако при статическом анализе доступны только сами скрипты, и весь процесс создания структуры базы данных должен быть проделан отдельно от самой базы. Это невероятно трудоемко, потому что придется полностью реализовывать логику работы базы данных, но *однако данный* этот подход имеет и свои преимущества. Так, например, каждая миграция будет разбираться как последовательность атомарных действий, и все изменения будут легко отслеживаемыми.
3. Динамический анализ скриптов миграции, в свою очередь, требует наличия рабочей базы данных. При таком подходе каждый скрипт представляет собой «черный ящик», то есть не происходит никакого анализа внутреннего содержимого скрипта, анализируется только изменение состояния базы данных до и после миграции. Таким образом, динамический анализ избавляет от необходимости проделывать ту же работу по созданию схемы данных, которая происходит в самой базе. *Однако* данный подход имеет свои недостатки: из-за того, *есть всего 2 состояния:* что имеется всего два состояния базы *данных* – до и после миграции, *-* локализация ошибок может быть затруднена.
4. Одной из основных причин ошибок применения миграции *является* неправильная манипуляция данными, Это может произойти, например, при добавлении ограничения NOT NULL для столбца таблицы, который ранее не был проверен на наличие записей со значением NULL. *Такая ситуация/иллюсттрирует* представляет собой простой пример возможных ошибок, которые могут возникнуть в процессе миграции. Если же мы говорим о более сложных изменениях в структуре базы данных, таких, как добавление новых таблиц и разделение существующих таблиц с последующим перемещением данных, то обнаружение ошибок будет намного сложнее: для проверки потребуются специфичные данные, которые должны находиться в базе до применения миграции.
5. Несмотря на то, что успешное применение скрипта миграций может показаться достаточным условием корректности миграции и сохранения целостности данных, на деле это не всегда так. *При работе с данными* всегда существует определенный риск потери информации или ее неправильного изменения, но такие ошибки практически невозможно отследить без ручной проверки.
6. Важно отметить, что не все ограничения должны быть явно описаны в самой структуре, некоторые обеспечиваются работой приложения, использующего эту базу данных. Например, добавление ограничения NOT NULL для столбца не обязательно вызовет ошибку на реальной базе, если приложение, работающее не добавляло NULL записи.
7. Объединяя все сказанное, можно сформулировать требования для системы автоматического тестирования версионных миграций: система должна выявлять потенциальные проблемы в процессе применения миграций при помощи динамического анализа скриптов с использованием экземпляра базы данных. --длинновато

# Метод решения

1. Для выявления ошибок миграции, связанных с нарушением статического арбитража, необходимо наличие данных в базе на момент миграции. -- возможные нарушения нормализации.
2. *Не знаю как лучше обосновать то, что таблицы должны быть заполнены данными. Может быть стоит расписать более подробно, зачем в базе должны присутствовать данные на момент применения миграции?*
3. Однако возникает проблема генерации таких данных при отсутствии реверс-инжиниринга базы данных. Решением может быть генерация данных, основанная на информации, полученной из служебных таблиц PostgreSQL, содержащихся в служебных схемах и таблицах, таких как pg\_class, pg\_enum, pg\_catalog и многих других.

Однако, даже при наличии всей необходимой информации о находящихся в базе сущностях, могут возникнуть трудности с генерацией корректных данных. В основном сложности вызывает динамический арбитраж, который может устанавливать динамические ограничения для данных, а так же изменять их. В контексте PostgreSQL возможны следующие динамические ограничения:

1. Триггеры (triggers) - это функции, которые вызываются автоматически при определенных событиях, таких как вставка, обновление или удаление строк из таблицы. В теле функции можно определить логику проверки и применения ограничений. Например, можно создать триггер, который запрещает вставку строк с определенным значением в колонке:

CREATE FUNCTION check\_value() RETURNS TRIGGER AS $$

BEGIN

IF NEW.column\_name = 'forbidden\_value' THEN

RAISE EXCEPTION 'Value not allowed';

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER check\_value\_trigger BEFORE INSERT OR UPDATE ON table\_name

FOR EACH ROW EXECUTE PROCEDURE check\_value();

1. Правила (rules) - это объекты базы данных, которые позволяют перехватывать запросы к таблице и изменять их результат. Например, можно создать правило, которое изменит запрос на вставку строк таким образом, чтобы он применял определенные ограничения:

CREATE RULE insert\_rule AS ON INSERT TO table\_name

WHERE (NEW.column\_name = 'forbidden\_value')

DO INSTEAD

RAISE EXCEPTION 'Value not allowed';

1. Генерируемые колонки (generated columns) - это колонки, значения которых вычисляются автоматически на основе значений других колонок в таблице. Например, можно создать генерируемую колонку, которая будет содержать результат проверки значения другой колонки:

CREATE TABLE table\_name (

column1 INTEGER,

column2 TEXT,

is\_value\_allowed BOOLEAN GENERATED ALWAYS AS (column2 <> 'forbidden\_value') STORED

);

Эти способы могут использоваться вместе или отдельно для добавления динамических ограничений в PostgreSQL. Чтобы проверить логику таких динамических ограничений нужно полностью повторить логику работы базы данных, потому что в теле триггера может содержаться практически любой SQL код, который может обращаться к другим таблицам и даже создавать новые таблицы. Правила (rules) и вовсе позволяют подменять результаты SELECT запросов.

1. Статический арбитраж тоже усложняет процесс генерации корректных данных, требуя наличия специфических ограничений, например условие CHECK позволяет задавать произвольные условия для проверки значений отдельных колонок или целой одной записи.
2. *Условие CHECK намного проще чем триггер, но я не знаю каким образом можно выделить подмножество возможных значений, подходящих под условие. Если учитывать, что в условии CHECK можно использовать внутренние функции PostgreSQL, то оно не сильно отличается по своей функциональности от триггера, хотя номинально относится к статическому арбитражу.*
3. Таким образом, генерация и последующая вставка данных *является* нетривиальной задачей из-за множества факторов, связанных с внутренней логикой работы самой базы данных. Также нужно учитывать, что сгенерированные данные могут быть *неконсистентными — что такое, описать* по сравнению с теми данными, которые на самом деле будут находиться в базе вследствие работы приложения, и в этом случае проблема намного глубже, потому что повторение внутренней логики работы приложения не представляется возможным, если исходить только из скриптов миграций.

# Генерация данных

Рассматривая возможность автоматической генерации данных, можно прийти к выводу, что в общем случае невозможно создать абсолютно корректные искусственные данные. Это связано с тем, что клиентское приложение, которое работает с базой данных, контролирует добавление данных.

Однако, если отбросить идею создания абсолютно корректных данных, можно ограничиться только генерацией данных, но не их вставкой в таблицы. Такой способ позволит устранять ошибки в сгенерированных данных и не будет ограничивать разработчика в добавлении и изменении предложенных искусственных данных. *В таком случае* возможно/есть риск добавление изначально некорректных данных, но подобную ошибку может обнаружить только человек, который имеет доступ к структуре базы и представление о том, как клиентское приложение должно с ней взаимодействовать. В таком случае только ручная проверка данных может обнаружить некорректные данные, не предусмотренные логикой работы приложения.

## Порядок вставки генерируемых данных

Существующие связи между таблицами не позволяют заполнять их в произвольном порядке, так как индексы гарантируют, что в базе не существует связей, которые ссылаются на ещё не добавленные данные. В общем виде связи между таблицами представляют собой направленный мультиграф, где направление обозначает связь между родительской и дочерней таблицей. То есть при заполнении таблиц нужно учитывать множественные связи между таблицами и циклически связанные таблицы.

### Циклически связанные таблицы

Нарушения нормальных форм — переписать.

Важно отметить, что цикл из таблиц, связанных соотношением 1:N является вырожденным, так как в такой цикл невозможно добавить новые данные, если все таблицы изначально пустые. Работа с циклически связанными таблицами затруднительна и скорее всего является ошибкой проектирования.

Для демонстрации возможности создания циклически связанных таблиц со связями 1:N, можно рассмотреть следующий SQL-код:

CREATE TABLE a(id INTEGER PRIMARY KEY);

CREATE TABLE b(id INTEGER PRIMARY KEY);

INSERT INTO b (id) VALUES (1);

INSERT INTO a (id) VALUES (1);

ALTER TABLE a ADD COLUMN val\_a INTEGER NOT NULL REFERENCES b(id) DEFAULT 1;

ALTER TABLE b ADD COLUMN val\_b INTEGER NOT NULL REFERENCES a(id) DEFAULT 1;

Даже если в цикле есть хотя бы одна связь 0-1:N, где внешний ключ может быть NULL, целесообразность использования цикла все равно весьма сомнительна.

Таким образом я пришел к выводу что не нужно генерировать данные для циклически связанных таблиц — это не имеет смысла, потому что для взаимодействия с каждым циклом нужна своя уникальная логика.

### Вставка данных без учета циклов

Если не рассматривать циклически связанные таблицы, то можно определить порядок обхода вершин. Можно использовать алгоритм обхода, похожий на поиск в глубину. Различие с оригинальным алгоритмом будет заключаться в том, что на последующей итерации не всегда будет выбираться смежная вершина. На каждой итерации алгоритма будет выбираться вершина, которая не имеет зависимостей от ещё не пройденных вершин. Таким образом, можно определить порядок заполнения таблиц при условии отсутствия циклических связей.

## Принцип генерации данных через определение набора требований

В общем случае полное множество состояний базы данных невозможно рассмотреть. Тогда для ограничения объема генерируемых данных следует определить набор требований, которым должны удовлетворять генерируемые данные. Это поможет учесть возможные состояния базы, вызывающие типичные ошибки. Например, для колонки типа REAL возможны значения Infinity, -Infinity, NaN, и если при обработке такой колонки не проверяются эти особые значения, то может возникнуть ошибка. Также другие типы данных имеют свои уникальные значения, требующие проверки. Для колонок с возможными NULL значениями лучше добавить NULL значение, чтобы проверить поведение скрипта миграции.

Так же может быть необходимым добавление более сложных требований, например наличие дублирующих записей для колонок, не содержащихся в UNIQUE индексе или требование на минимальное/максимальное количество генерируемых записей для таблицы.

Таким образом необходимо иметь возможность контролировать процесс генерации записей и расширять список требований для отдельных колонок и таблиц.

Рассмотрим следующую таблицу:

CREATE TABLE persons (

id SERIAL PRIMARY KEY,

name VARCHAR(50) UNIQUE NOT NULL,

age INTEGER,

is\_avaliable BOOLEAN NOT NULL

);

Базовый список требований для неё будет выглядеть следующим образом:

1. Проверка поля name на наличие пустой строки и строки длины 50.

(2 записи)

1. Проверка поля age на наличие значений 0, 1, -1, MAXINT, MININT, NULL.  
   (6 записей)
2. Проверка поля is\_available на наличие значений True, False.

(2 записи)

1. Проверка на наличие записи, дублирующей значения колонок age и is\_available, так как они не содержатся в UNIQUE индексе.

(1 запись)

На основании этих требований можно сгенерировать от 6 частичных записей (если совмещать в одной записи требования для разных колонок) до 10 частичных записей (если не совмещать записи с требованиями). Под словами «частичная запись» подразумевается, что известны значения только нескольких колонок, например при проверке поля name известно только значение этой колонки, а значения всех остальных колонок следует сгенерировать.

Таким образом, список требований будет формировать список частичных записей, которые будут дополняться генерируемыми данными и преобразовываться в SQL скрипт добавления данных.

### Определение множества значений для генерируемых данных

**домен**

При генерации значений столбца необходимо учитывать не только специфичные значения, но и более «обычные» варианты. Случайная генерация может привести к нежелательным результатам, поэтому лучше использовать заранее приготовленное множество значений для конкретного типа данных, либо итеративный подход для получения следующего значения.

Например, для колонки типа INTEGER с возможными NULL значениями будет определен следующий порядок генерируемых значений: NULL, 0, 1, -1, 2, -2, … . Для практически всех базовых типов postgresql можно применять итеративный подход для получения следующего допустимого значения. Однако для некоторых типов, таких как JSON, XML, VARCHAR, BYTEA, TEXT, CIDR, ARRAY, RANGE, DOMAIN, и нескольких других, нужно заранее определять множество допустимых значений, например с помощью конфигурационного файла.

В PostgreSQL также возможно добавление пользовательских типов, таких как ENUM, COMPOSITE и основанных на типах DOMAIN и RANGE, из которых только для ENUM можно использовать все возможные значения типа. Более того, можно добавить требование, чтобы все значения ENUM присутствовали в таблице.

### Ограничение CHECK

В настоящее время я не рассматриваю обработку ограничения CHECK, поскольку оно проявляет поведение, более близкое к динамическому арбитражу, чем к статическим ограничениям. Если в условии используется ограничение CHECK, то слишком сложно выделить множество возможных значений, удовлетворяющих этому условию. Кроме того, такое условие может обращаться сразу к нескольким колонкам, что также увеличивает сложность его обработки.

## Генерация данных на основе шаблонных процессоров

Для каждой таблицы в PosttgreSQL можно получить полное описание её колонок, индексов, ограничений и внешних связей. На основе этих данных будет составлен базовый список требований к данным, генерируемым для каждой обрабатываемой таблицы. Этот список требований можно представить в виде составного шаблона для генерации записей.

Рассмотрим следующую таблицу:

CREATE TABLE users (

id SERIAL PRIMARY KEY,

name VARCHAR(50) NOT NULL,

age INT,

email VARCHAR(100) UNIQUE,

phone CHAR(10),

is\_active BOOLEAN DEFAULT true,

created\_at TIMESTAMP DEFAULT NOW(),

updated\_at TIMESTAMP,

address TEXT,

location\_id INTEGER REFERENCES location(id)

);

Используя golang templates можно создать следующий шаблон, результатом обработки которого будет список записей, готовых для вставки в исходную таблицу:

*Я говорю про golang templates, потому что саму программу буду реализовывать именно на языке go, этот язык наиболее привычен для меня. Не принципиально использовать именно встроенный в язык golang шаблонный процессор, просто это хорошо знакомый мне инструмент.*

{{check(name).values(«», makestrlen(50))}} — проверка на наличие записей

{{check(age).values(0,1,-1,maxint,minint)}} — проверка на наличие записей

{{check(is\_active).all()}} — проверка на наличие всех записей

{{check(phone).values(«»,makestrlen(10))}} — проверка на наличие записей

{{check(created\_at).values(default,now(), now().add(hour), now().sub(hour))}} — проверка на наличие записей. В том числе проверка с добавлением DEFAULT значения, которое заполняет база данных если не было передано конкретное значение

{{check(updated\_at).values(now(), now().add(hour), now().sub(hour))}} — проверка на наличие записей

{{check(address).values(«», makestrlen(1024)}} — проверка на наличие записей

{{check(localtion\_id).all()}} — проверка на наличие всех возможных записей, учитывая записи, уже сгенерированные для таблицы location

{{check(age,email,phone,is\_active,created\_at,updated\_at,address,location\_id).nulls()}} — проверка на наличие null записей в столбцах, где возможно null значение

{{uniq(email)}} — добавление дублирующей записи для всех столбцов кроме id и email, которые содержатся в UNIQUE индексе

Используя язык шаблонов можно задавать произвольные условия, используя особые функции языка шаблонов. Например для следующей таблицы нужно добавить требование на наличие записей, в которых для одного ребёнка указаны два разных родителя:

CREATE TABLE peoples (

id SERIAL PRIMARY KEY

);

CREATE TABLE relations (

id SERIAL PRIMARY KEY,

parent\_id INTEGER NOT NULL REFERENCES peoples(id),

child\_id INTEGER NOT NULL REFERENCES peoples(id),

UNIQUE(parent\_id, child\_id)

);

Шаблон с указанным условием будет выглядеть следующим образом:

{{$parent\_1 := anyof(peoples.id) }} -- получение любого id из таблицы peoples

{{$parent\_2 := anyof(peoples.id, except($parent\_1)) }} -- получение id из таблицы peoples, который не равен parent\_1

{{$child := anyof(peoples.id, except($parent\_1, $parent\_2)) }} -- получение получение id из таблицы peoples, который не равен parent\_1 и parent\_2

{{check(parent\_id, child\_id).values($parent\_1, $child)}}

{{check(parent\_id, child\_id).values($parent\_2, $child)}}

*Это лишь пример, можно сделать более привлекательный внешний вид для шаблонов.*

Язык шаблонов позволяет разработчику указать произвольные условия для генерируемых данных, что позволяет тонко контролировать процесс генерации данных.

## Этапы генерации данных

На данный момент генерация данных будет происходить в 2 этапа:

1. Генерация базового шаблона для таблицы, либо загрузка модифицированного шаблона из файла.
2. Генерация записей по шаблону.

Так же на каждом этапе возможна модификация данных для поддержания консистентности, либо для добавления специфичных данных.

# Система автоматического тестирования версионных миграций

## Основные этапы работы программы

Почитать про аавтоматическое стестироване вобщем случае

Работа программы будет разделена на несколько этапов, это сделано с целью повышения контроля процесса работы программы и упрощения взаимодействия с пользователем.

1. Аггрегация информации и генерация базовых шаблонов.
   1. Получение информации о таблицах и связях из базы данных.
   2. Построение графа связей таблиц.
   3. Проверка графа связей на наличие циклов.
   4. Генерация базовых шаблонов для таблиц.
2. Генерация данных.
   1. Загрузка графа связей таблиц и информации о таблицах

(возможно кеширование).

* 1. Загрузка шаблонов генерации для таблиц.
  2. Генерация записей.
  3. Генерация SQL скриптов вставки данных.

1. Применение миграции следующей версии.

Стоит отметить, что при генерации данных возможна ситуация когда невозможно добавление достаточного количества записей в таблицу. Такая ситуация может возникнуть из-за ограничений, существующих для этой таблицы. Например если таблица содержит внешний ключ с ограничением UNIQUE NOT NULL, то для добавления достаточного количества записей нужно определенное количество записей в родительской таблице. В таких случаях нужно выводить сообщение о невозможности создания новых записей и продолжить генерацию для оставшихся таблиц.

Также для генерации данных стоит предоставлять особые настройки, например, исключение таблицы из генерации. Это означает, что будут использоваться только те данные, которые уже записаны в эту таблицу (или таблица пустая и данных нет). Также можно полностью исключить таблицу из графа связей с отображением тех таблиц, которые также будут исключены из-за зависимостей от указанной таблицы.