## Введение

Для работы многих веб-приложений необходимо хранилище данных. Часто для хранения этих данных используются реляционные базы данных, например PostgreSQL. В реляционных базах данные хранятся структурировано, а структуры описываются схемами. Большие веб-приложения меняются со временем, и соответственно изменяются схемы в реляционных базах данных. Изменение схем часто реализуют при помощи версионных миграций, которые представляют собой версионированную последовательность из нескольких SQL скриптов. При применении этих скриптов могут возникать ошибки. Чтобы отследить некоторые из них достаточно применить миграции на пустой базе, но так же возможны ошибки связанные не с миграцией схемы, а с миграцией данных. В основном эти ошибки будут возникать из за нарушения CONSTRAINT-а.

CONSTRAINT это ограничения для столбцов и таблиц, которые дают возможность проверять корректность данных на этапе записи в базу. Такими ограничениями являются вторичные ключи таблиц, так как они не позволят создать запись которая ссылается на несуществующие данные в другой таблице. Типы колонок так же являются ограничениями, например нельзя записать строку в колонку, предназначенную для типа INTEGER. Но ограничения типов слишком грубые и дополнительные CONSTRAINT-ы позволяют дополнять их.

## Постановка задачи

Есть несколько подходов для отслеживания ошибок миграций, например статический и динамически анализ скриптов миграций. При статическом анализе нужен только SQL код скриптов для поиска ошибок. Для динамического же помимо самих скриптов понадобится рабочая база данных, на которой эти скрипты будут применяться. То есть при динамическом анализе не нужно знать что находится в скриптах миграций, достаточно только знать их версии. При статическом анализе доступны только скрипты, и всю логику по созданию схемы, которую делает база, придётся создавать отдельно от базы. Это невероятно трудоёмко, однако при статическом подходе есть и преимущество – каждая миграция будет разбираться пошагово, а значит все изменения будет легко отследить. В отличие от динамического анализа, при котором имеется лишь 2 состояния базы – до и после миграции. Вариант с динамическим анализом имеет свои недостатки, но статический анализ потребует намного больше усилий в реализации при том что он не сможет полностью повторить логику работы базы, и как следствие корректно тестировать миграции. Динамический анализ намного более привлекателен тем, что тестируется не логика миграции, а её результат. Достаточно лишь создать все возможные ситуации, при которых миграция может вызвать ошибку.

Как упоминалось ранее основная причина ошибок – нарушение CONSTRAINT-а. Наиболее простой пример – добавление NOT NULL ограничения для столбца. Тогда если в таблице существует запись с NULL значением столбца, то миграция завершится с ошибкой. Решением будет добавление значение DEFAULT к ограничению NOT NULL, тогда все NULL элементы получат указанное дефолтное значение. Это простейший пример, затрагивающий всего лишь один столбец в одной таблице, если же схема изменяется более существенно, добавляя и разделяя таблицы, то ошибки будут намного сложнее, так как для их определения необходима схема, заполненная определёнными данными. Более того, даже успешное применение миграций не гарантирует сохранность данных.

Все описанное выше больше относится к миграциям, которые повышают версию базы (миграции «вверх»), но также бывает необходимо откатить изменения с новой версии обратно на старую. Для этого существуют миграции «вниз», к которым применимы те же требования, что и к миграциям «вверх». То есть они должны без ошибок изменять схему и гарантировать миграцию данных. Это означает то что миграции «вверх» и «вниз» должны быть биективны друг относительно друга. Если это условие выполняется, то возможно проверять не только корректность миграции схемы, но и корректность миграции данных.

Объединяя все сказанное выше можно сформулировать требование к системе автоматического тестирования: система должна обнаруживать возможные нарушения CONSTRAINT-ов в версионных миграциях с исполь-зованием динамического анализа.

Так же нельзя игнорировать то что не всегда реальные ограничения явно описаны в самой схеме, они могут обеспечиваться работой приложения, использующего базу. Тогда изменение типа столбца TEXT → VARCHAR(20) не будет вызывать ошибку на реальной базе. Так что желательно иметь возможность указать внешние (неявные) ограничения.

## Метод решения

Для обнаружения ошибок, связанных с миграцией данных необходимо сначала заполнить базу. Учитывая связи между таблицами нельзя добавлять данные в случайном порядке, так как CONSTRAINT-ы связей не позволят добавлять некорректные данные, которые ссылаются на ещё не добавленные записи в других таблицах. То есть нужен определённый порядок вставки записей в таблицы. Чтобы определить порядок необходимо иметь представление о схеме данных. Все данные о таблицах и их связях PostgreSQL позволяет получить при помощи служебных таблиц. Если отобразить все связи между таблицами, то они будут представлять собой направленный мультиграф. Это значит что возможны множественные связи между двумя таблицами и циклически связанные таблицы. Циклические зависимости особенно усложняют вставку данных, так записи должны добавляться в таблицы в определённом порядке. Стоит отметить что в изначально пустые циклически связанные таблицы невозможно добавить данные, если связи между ними представляют собой 1:N связь. Но если в цикле есть хотя бы одна связь 0-1:N, то можно добавить данные в одну таблицу, а значит и во все остальные дальше по циклу. Работа с циклически связанными таблицами затруднительна и возникновение цикла чаще всего связано с ошибкой проектирования, но существование циклической связи вполне возможно.

Описание всех связей между таблицами действительно создаёт мультиграф, но в предыдущем абзаце сказано что не все связи являются жёсткими, некоторые связи опциональны. Тогда если предположить что в схеме нет жёсткого цикла из таблиц, то структура связей будет больше похоже на дерево. А в дереве всегда можно найти корневую вершину, с которой можно начать заполнение таблиц.

## Эффективность метода

Если гарантировать что для каждой связи между таблицами будет создана хотя бы одна запись, то можно обнаружить ошибки CONSTRAINT-ов при изменении связей таблиц. В худшем случае каждая таблица связана со всеми остальными и тогда для каждой связи в ней будет ровно записей. Тогда для всех таблиц будет записей. То есть получается ассимптотика, где – количество таблиц или , где – количество связей между таблицами.

## Генерация данных

Как говорилось выше — для проверки корректности миграции необходимо чтобы в таблицах содержались данные. То есть для каждой миграции должен быть некоторый набор записей. Но при создании этих записей вручную можно ошибиться и пропустить какую-нибудь связь, или не добавить NULL значение для NULL-able колонки. Чтобы немного улучшить ситуацию, данные для таблиц можно сгенерировать. Действительно, не всегда для проверки миграции нужны определённые значения в записях, в большинстве случаев можно использовать случайные данные, так как важно лишь само наличие записи.

К сожалению генерировать абсолютно корректные данные очень сложно. Например, если у таблицы есть триггер, который производит поиск в другой таблице, то генератор не сможет понять что сначала надо добавлять данные в другую таблицу.

Таким образом генерация может помочь заполнять таблицы, но полагаться на неё полностью не стоит.

## Вставка данных

Предполагается, что миграция происходит на схеме, в которой уже содержатся данные. То есть необходимо перейти от состояния с пустой базой до состояния с заполненной. Самый логичный способ вставить данные это сделать INSERT, но у такого варианта сразу появляется недостаток. Этот недостаток заключается в динамическом арбитраже. В случае с PostgreSQL к динамическому арбитражу относятся триггеры и генерируемые колонки. В рамках задачи желательно не проверять корректность вставки данных, так как это несколько другая задача.

Существует два варианта записать данные в таблицу игнорируя динамический арбитраж:

1. Сделать дамп базы и пересоздать её с уже заполненными таблицами.
2. Отключить триггеры для таблицы средствами PostgreSQL. Для генерируемых колонок данные можно вставлять явно, либо позволить отработать процедуре генерации, так как доступа к другим таблицам не будет.

#### Заполнение через дамп

Существует утилита pg\_dump, которую используют для бекапов и для переноса базы с машины на машину. Результат работы утилиты — SQL скрипт, состоящий из трёх частей:

1. Создание таблиц и счётчиков (SEQUENCE).
2. Запись данных.
3. Создание ограничений, индексов и триггеров.

Если применить скрипт на той же базе, то он будет пересоздавать все элементы. Таким образом на втором шаге будут созданы все таблицы, но у них не будет никаких связей, и у колонок будут отсутствовать какие-либо ограничения. Отсутствие связей позволит игнорировать порядок вставки в таблицы, так как индексы будут создаваться позже. Но в том же и недостаток — случайно генерируемые данные могут не подходить под ограничения, и тогда может возникнуть ошибка, которая не имеет к миграции никакого отношения.

#### Заполнение через отключение триггеров

Отключив триггеры можно сымитировать состояние работающей базы. Последовательная вставка в таблицы позволит легко отслеживать ошибки в генерируемых данных, и не будет проблем с генерируемыми столбцами. Минусы этого подхода не столь значимы: циклически связанные таблицы заполнить таким образом невозможно. Но, как говорилось ранее, циклическая связь это скорее ошибка проектирования, которую надо уметь обнаруживать, но не обязательно проверять корректность работы.

### Проверка корректности ограничений

Имея возможность сгенерировать данные и записать их в базу стоит перейти к основной цели исследования — проверке выполнения ограничений при миграции.

Тут так же есть два варианта:

1. Проверить всё и сразу — заполнить все таблицы, создать запись для каждой связи и создать все возможные сочетания связей.
2. Создать минимальное количество записей чтобы покрыть все связи между таблицами, а затем изменять эти записи.

#### Проверка всеми данными

Полностью заполненная база будет полезна не только для проверки корректности миграций, но и для того чтобы тестировать запросы. Минусом будет только сложность генерации сильно связанных данных.

В рамках одной таблицы имеет смысл генерировать более одной записи, если у колонок есть специальные ограничения. Например в таблицу с NULL-able колонкой добавить NULL запись и не NULL запись, для колонки без ограничения по уникальности добавить две одинаковые записи и одну отличающуюся от них. Так

Если смотреть только на одну таблицу, то

При генерации всех данных сразу миграция прервётся на первой ошибке (если ошибка есть). Это значит что пока текущая ошибка не исправлена об остальных ошибках известно не будет.

Если же создать минимальное количество записей чтобы покрыть все связи, то можно проверять корректность миграции для каждого индекса по отдельности. Минимальное количество записей — это одна запись на одну связь. Тогда чтобы проверять ограничения, связанные с типом колонок достаточно изменить одну запись.

В некотором смысле статические ограничения можно проверять, изменяя всего лишь одну запись.

и проверять все арбитражи последовательно (изменяя ), то значительно возрастёт время проверки (потому что постоянно будет применяться один и тот же скрипт).

Генерировать данные для таблиц это не самая простая задача, так как при заполнении таблиц случайными данными могут возникнуть проблемы.

Например, приложение, работающее с базой, добавляет по одной записи на каждого пользователя, но при этом колонка user\_name не содержит контракт на уникальность которые невозможны при работе приложения с базой