## Введение

1. Хранилища данных являются 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.
2. Однако требования к приложению могут изменяться со временем, что приводит к изменению структуры базы данных, которую они используют. Версионные миграции представляют практичный и распространенный подход к управлению структурами баз данных.

##### Что такое миграции?

1. Каждая миграция – это скрипт, который изменяет версию схемы данных. Обычно каждый скрипт соответствует определенной версии приложения, которое работает с этой базой данных и требует определенной структуры данных. Миграции позволяют управлять изменением структуры базы данных вместе с изменениями в приложении.

##### Риски, связанные с миграциями

1. Хотя миграции представляют собой практичный и распространенный подход к управлению структурами баз данных, они также подвержены ошибкам. Некоторые из этих ошибок могут быть отслежены путем применения миграции ~~на~~ к базе, которая не содержит никаких данных, кроме тех что были созданы при предыдущей миграции.
2. Однако существуют ошибки, связанные с изменением статического арбитража, которые невозможно проверить при отсутствии данных в таблицах. Статический арбитраж – это набор проверок и ограничений, применяемых к данным при их добавлении, модификации и удалении. Он включает в себя такие элементы, как вторичные ключи, типы колонок и их атрибуты. Эти ограничения могут гарантировать правильность данных и обеспечить их корректное использование.
3. Поэтому ошибки, связанные с изменением статического арбитража или некорректным представлением о консистентности уже существующих данных, могут быть наиболее сложными для отслеживания. При работе с версионными миграциями необходимо учитывать эти риски и тщательно проверять каждый скрипт для обеспечения сохранения целостности и достоверности данных.

## Постановка задачи

1. Для отслеживания ошибок, возникающих в процессе миграций баз данных, существуют различные подходы, например статический и динамический анализ скриптов миграции.
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 данные 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, однако данный подход имеет и свои преимущества. Так, например, каждая миграция будет разбираться пошагово, и все изменения будут легко отследить.
3. Динамический анализ скриптов миграции, в свою очередь, требует наличия рабочей базы данных. При этом достаточно знать только версии скриптов миграции, без необходимости анализировать их содержимое. Таким образом, динамический анализ избавляет от необходимости проделывать ту же работу по созданию схемы данных, которую делает сама база. Однако данный подход имеет свои недостатки: из-за того, 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 была проверена на наличие записей со значением NULL. Такая ситуация 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 и разделение таблиц, то обнаружение ошибок будет намного сложнее, так как для проверки потребуются специфичные данные, которые будут находиться в базе до применения миграции.
5. Несмотря на то, что успешное применение миграций может показаться достаточным для обеспечения корректности и 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. Помимо миграций "вверх", которые повышают версию базы данных, также существуют миграции "вниз", которые понижают версию. Требования к миграциям "вниз" те же, что и к миграциям "вверх". Они должны корректно изменять схему и гарантировать миграцию данных. Если миграции "вверх" и "вниз" для одной и той же версии структуры базы данных биективны относительно друг друга, то есть возможность проверять как корректность миграции схемы, так и корректность миграции данных.
7. Важно отметить, что не все ограничения явно описаны в самой структуре базы данных, некоторые из них обеспечиваются работой приложения, использующего базу данных. Например, добавление ограничения NOT NULL для столбца не обязательно вызовет ошибку на реальной базе.
8. Объединяя все вышесказанное, 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.

## Метод решения

1. Для выявления ошибок миграции, связанных с некорректным изменением статического арбитража, необходимо наличие данных в базе на момент миграции. Однако возникает проблема генерации таких данных при отсутствии реверс-инжиниринга базы данных. Решением может быть генерация данных исходя из описания таблиц, колонок и связей, но данный способ имеет свои сложности.
2. Например, в случае динамического арбитража, который часто реализуется при помощи триггеров и генерируемых колонок, необходимо проверить всю логику работы триггера, который может содержать в себе практически любые SQL команды. Следовательно, чтобы проверить условия для добавления данных, нужно полностью повторить логику работы базы данных. Следует заметить, что динамический арбитраж - это только один из ярких примеров. Статический арбитраж также может усложнить процесс генерации корректных данных, требуя наличия специфических ограничений, например если таблицы циклически связаны индексами.
3. Таким образом, генерация и последующая вставка данных является задачей нетривиальной из-за множества факторов, связанных с внутренней логикой работы самой базы данных. Также сгенерированные данные могут быть неконсистентными по сравнению с теми данными, которые на самом деле будут находиться в базе вследствие работы приложения, и в этом случае проблема намного глубже. Так что вместо генерации данных будет намного более предпочтительным вариантом (1), если для тестирования миграций будут использоваться отдельные скрипты, которые будут заполнять базу в соответствии с её текущей структурой. Этот метод является более эффективным, потому что изначально в базу не будут добавлены данные, которые не предусмотрены логикой работы приложения. Более того, имея доступ к базе данных до и после миграции можно использовать внутренние служебные таблицы PostgreSQL для обнаружения неиспользуемых связей и неполноты охвата множества возможных значений для отдельных колонок.

## Подсчет общего количества данных в связанных таблицах

1. Если все возможные связи таблиц будут иметь записи, то количество общих записей составит примерно O(n^2), где n является количеством связанных таблиц.
2. Кроме того, каждая из возможных связей таблиц должна содержать особые подмножества значений для отдельных колонок. Например, для колонки без ограничения NOT NULL должна быть предусмотрена запись с NULL значением. Также для REAL-колонки значения должны включать +INFINITY, -INFINITY, NaN.
   1. Таким образом, общее количество данных с учетом возможных значений отдельных колонок будет составлять , где m — количество колонок в таблице, а k — количество определенных контрольных значений из множества значений конкретной колонки. На практике k — потенциально небольшая постоянная.

## Проверка заполненности базы

В PostgreSQL служебные таблицы предоставляют различную информацию о базе данных, такую как информация о таблицах, индексах, связях таблиц, типах колонок и других статических ограничениях.

Имея доступ к этой информации и к данным, находящимся в проверяемых таблицах, можно простым обходом всех записей проверять существование всех возможных связей между таблицами, а также проверить возможные значения отдельных колонок на наличие уникальных и краевых значений.

Основываясь на результатах анализа данных текущей структуры таблиц и связей, можно сгенерировать отчет об охвате возможных комбинаций связей данных и их значений. Результат анализа можно использовать для добавления новых данных и увеличения охвата возможных ситуаций изменения этих данных.

======

Полагаю, тема вполне серьёзная и актуальная. Действительно, из-за возможной некорректности базы данных после очередного изменения структуры возникает риск ошибки в приложении, и если оно критически важное, потери могут быть слишком велики. Так что игра стоит свеч, пусть хоть это будет.

Далее, то, что Вы говорите об общем количестве данных, обсуждается при исследовании программ при их тестировании. Количество вариантов получается слишком большим, и без автоматической генерации не обойтись. Надо только пояснить, что и по какой причине Вы генерируете.

Что касается Вашего предложения (1), я его понимаю как предложение заполнить БД заведомо корректными данными и на них проверить работу. Для БД, в отличие от программ, это обосновано, но не в случае изменения структуры. Здесь может быть что угодно. Но мы знаем, что будет изменено, и будем действовать целенаправленно.

Возможно, я что-то неправильно понял: текст слишком короткий, но считаю дальнейшее продвижение и обсуждение в любом случае полезным.

Рассмотрев возможность автоматической генерации данных, я пришел 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. Следовательно, для автоматической генерации искусственных данных необходимо разработать специальное приложение, которое содержит всю необходимую логику работы с базой, подобную той, которая есть в клиентском приложении.

Однако, если отбросить идею создания абсолютно корректных данных, можно ограничиться созданием SQL-скриптов. Такие скрипты будут содержать предположительно верные команды для добавления данных. Этот метод значительно упростит создание искусственно заполненной базы и не ограничит разработчика в изменении предложенных данных.

Если использовать такой подход, то он неплохо соотносится с моим предложением об использовании скриптов для заполнения базы. Хотя в таком случае возможно 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 валидация данных может обнаружить некорректные данные, которые не предусмотрены логикой приложения.

Касательно Ваших слов «*Но мы знаем, что будет изменено, и будем действовать целенаправленно*». Я не исследую сам скрипт миграции, потому что он может содержать сложно анализируемую логику, которую весьма проблематично обработать автоматически.

Я основываю свою идею проверки миграций на концепции черного ящика: даже без конкретной информации о предстоящих изменениях можно предположить, какие из этих изменений могут вызвать ошибку. Например, добавление UNIQUE или NOT NULL ограничений без предварительной проверки может вызвать ошибку. Либо изменение типа колонки с TEXT на VARCHAR(255) без проверки на максимальный размер уже находящихся в таблице записей.

*Не знаю как лучше обосновать то, что таблицы должны быть заполнены данными.*

### Вставка 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 с одной или несколькими компонентами связности, 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

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 сомнительно.

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.

##### Заполнение 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.

##### Ограничение объема генерируемых данных

Ранее я неверно подсчитал общее количество записей, необходимых для покрытия всех возможных сочетаний связей. Общее количество записей в таком случае будет иметь ассимптотику , где m — количество записей во внешней таблице, n — количество внешних ключей в таблице, k — количество таблиц. Таким образом, необходимо минимизировать количество генерируемых записей.

##### Генерация данных для одной таблицы

Нужно определить набор условий для каждого из типов, учитывая атрибуты колонок и базовые статические ограничения (NOT NULL, UNIQUE). Совокупность набора условий для всех колонок таблицы будет являться набором условий для всей таблицы. Количество генерируемых записей в таблице будет равно максимальному количеству условий среди всех колонок этой таблицы. Если использовать такой подход, то количество записей будет минимальным, а их разнообразие максимальным.

И что делать с CHECK? - игнорировать. Несмотря на то что CHECK используется скорее как статическое ограничение, на деле он делает практически то же самое что и триггер. Конечно условие обрабатывать намного проще, но я все равно не смогу выделить множество возможных значений. Это скорее тема для отдельной работы.

Как тогда проверять UNIQUE для нескольких колонок? - костылями. Создать условную мапу из соответствий и для каждой созданной записи проверять есть ли она в мапе. Если есть, то перегенерировать строчку. (сначала полный перебор пула, потом рандомная генерация)

Видимо придется использовать рандом для генерации данных, можно даже использовать конкретный random.Seed чтобы результат повторялся (хотя бы частично). Но вообще я думал сделать пул значений для каждого из базовых типов.

А с пользовательскими типами что делать?

Enum, composite, range, domain, array. Тут проблема только с DOMAIN, потому что внутри него условие CHECK, которое я буду абьюзить. Остальные создаваемые типы могут быть только композицией базовых, а для всех базовых я генерацию сделаю. Для домейна пусть разрабы сами дают мне пул значений, а я буду только из него брать. Думаю что не нужно все таки делать рандомную генерацию. Скорее нужно указывать настройки для генерации пула значений. Для интов тупа +1, для времени интервал, для строк — хз, просто файл.

Думаю стоит соединить рандомную генерацию и пул значений, с приоритетом пула. Это позволит создавать искуственные данные, которые будут простыми для человеческого восприятия.

Доп генерация данных для родительских таблиц, если не хватило значений. Восходящая генерация с указанием количества необходимых данных. Добавление должно быть по типу транзакции — если не получилось добавить где-то сверху, то все добавление прерывается.