Экзаменационный проект
по дисциплине
Проектирование баз данных
студента гр. **M34361**
Деминцева Данила Дмитриевича

по теме

Сиситема управления доступами

Описание проекта

Проект представляет собой систему управления доступами. Основные сущности тут - сервис и команда. У каждого сервиса есть ответственная за него команда, члены которой могут выдавать доступы к сервису, за который они ответственны. Для обеспечения гранулярности доступов внутри сервиса, есть действия и секции. Действие - это некоторое обозначение того, что может сделать пользователь внутри сервиса - записать что-то, прочитать, удалить, выгрузить итд. Секции - это подразделы сервиса, имеющие иерархическую структуру и представляющие собой лес деревьев. Возможность делать действия в секции выдается на команду (потому что индивидуальные роли - зло), такая роль и является сущностью, ради которой всё затевалось.

Построение отношений

В результате предварительного проектирования были выделены следующие отношения:

- users — пользователи в системе.
- teams — команды в системе.
- team_members — показывает принадлежность человека команде. Человек может быть в нескольких командах одновременно (например, один database administrator на три команды).
- services — сервисы в системе, у каждого сервиса есть ответственная за него команда.
- actions — действия, которые можно делать в конкретном сервисе (например: read, write, create, delete).
- sections — раздел сервиса. Имеют иерархическую структуру, отсюда два следующих отношения.
- section_parents — отношение ребенок-родитель.
- section_root_paths — отношение вершина - путь до корня.
- section_codes — отдельно вынесенные строковые идентификаторы секций, уникальные в рамках сервиса. Мотивация такого вынесения приведена в комментарии к ERM.
- roles — роли - выданная команде возможность делать действия в секции (и ее потомках) с опциональной датой истечения.

Отношение users

Атрибуты:

- user_id — идентификатор пользователя
- login — уникальный логин пользователя
- name
- surname

Функциональные зависимости:

- user_id -> login, name, surname
- login -> user_id, name, surname

Ключи:

- user_id
- login

Нормализация:

- Отношение находится в 1НФ так как у него есть ключ, все атрибуты атомарны и нет повторяющихся групп
- Отношение находится в 2НФ так как все ключи простые, а значит, неключевые атрибуты зависят от ключей целиком
- Отношение находится в 3НФ так как каждый неключевой атрибут зависит напрямую от каждого из ключей
- Отношение находится в 5НФ по теореме Дейта-Фейгина, так как находится в 3НФ и все ключи простые

Отношение teams

Атрибуты:

- team_id — идентификатор команды
- name — название команды

Функциональные зависимости:

- team_id -> name

Ключи:

- team_id

Отношение team_members

Атрибуты:

- user_id — идентификатор пользователя
- team_id — идентификатор команды

Функциональные зависимости:

- user_id, team_id -> user_id, team_id

Ключи:

- user_id, team_id

Отношение services

Атрибуты:

- service_id — идентификатор сервиса
- code — уникальный код сервиса
- name — название сервиса
- team_id — идентификатор команды, ответственной за сервис

Функциональные зависимости:

- service_id -> code, name, team_id
- code -> service_id, name, team_id

Ключи:

- service_id
- code

Отношение actions

Атрибуты:

- action_id — идентификатор действия
- code — код действия, уникальный в рамках сервиса
- service_id — идентификатор сервиса, которому принадлежит действие

Функциональные зависимости:

- action_id -> code, service_id
- service_id, code -> action_id

Ключи:

- action_id
- service_id, code

Нормализация:

- Отношение находится в 1НФ так как у него есть ключ, все атрибуты атомарны и нет повторяющихся групп
- Отношение находится в 2НФ так как все ключи простые, а значит, неключевые атрибуты зависят от ключей целиком
- Отношение находится в 3НФ так как каждый неключевой атрибут зависит напрямую от каждого из ключей
- Отношение находится в 5НФ, поскольку декомпозировать его дальше не представляется возможным (корректная декомпозиция по service_id и code очевидно невозможна, а вот декомпозиция по action_id чуть сложнее, но при рассмотрении всевозможных наполнений декомпозированных отношений мы можем получить кортежи, которых не могло быть в исходной таблице (а именно, дубли code для одного service_id с разным action_id))

Отношение sections

Атрибуты:

- service_id — идентификатор сервиса, которому принадлежит секция
- code_id — идентификатор кода секции, уникального в рамках сервиса
- name — название секции

Функциональные зависимости:

- service_id, code_id -> section_id, name

Ключи:

- service_id, code_id

Отношение section_codes

Атрибуты:

- code_id — идентификатор кода
- code — значение кода

Функциональные зависимости:

- code_id -> code
- code -> code_id

Ключи:

- code_id
- code

Отношение section_parents

Атрибуты:

- service_id — идентификатор сервиса
- section_code_id — идентификатор кода секции
- parent_code_id — идентификатор кода родителя

Функциональные зависимости:

- service_id, section_code_id -> parent_code_id

Ключи:

- service_id, section_code_id

Отношение section_root_paths

Атрибуты:

- service_id — идентификатор сервиса
- section_code_id — идентификатор кода секции
- path_item_code_id — идентификатор кода секции на пути к корню

Функциональные зависимости:

- service_id, section_code_id, path_item_code_id -> service_id, section_code_id, path_item_code_id

Ключи:

- service_id, section_code_id, path_item_code_id

Отношение roles

Атрибуты:

- team_id — идентификатор команды.
- section_id — идентификатор секции.
- action_id — идентификатор действия
- granted_by — идентификатор пользователя, выдавшего роль.
- exiration_ts — опциональная дата истечения действия роли.

Функциональные зависимости:

- team_id, section_id, action_id -> granted_by, exiration_ts

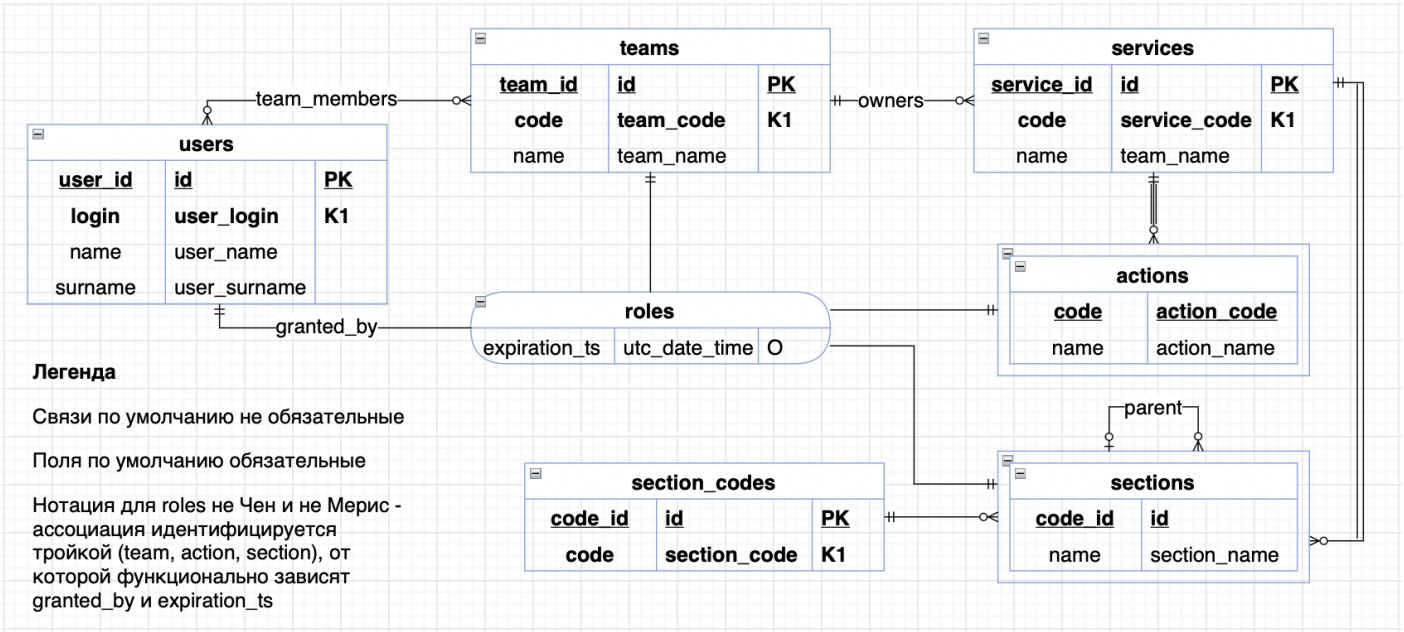
Ключи:

- team_id, section_id, action_id

Нормализация:

- Отношение находится в 1НФ так как у него есть ключ, все атрибуты атомарны и нет повторяющихся групп
- Отношение находится в 2НФ так как неключевые атрибуты зависят от ключа целиком
- Отношение находится в 3НФ так как каждый неключевой атрибут зависит напрямую от ключа
- Отношение находится в 5НФ так как корректная дальнейшая декомпозиция возможна только по ключу, а разбиение на разные таблицы granted_by и expiration_ts нам ничего не дает.

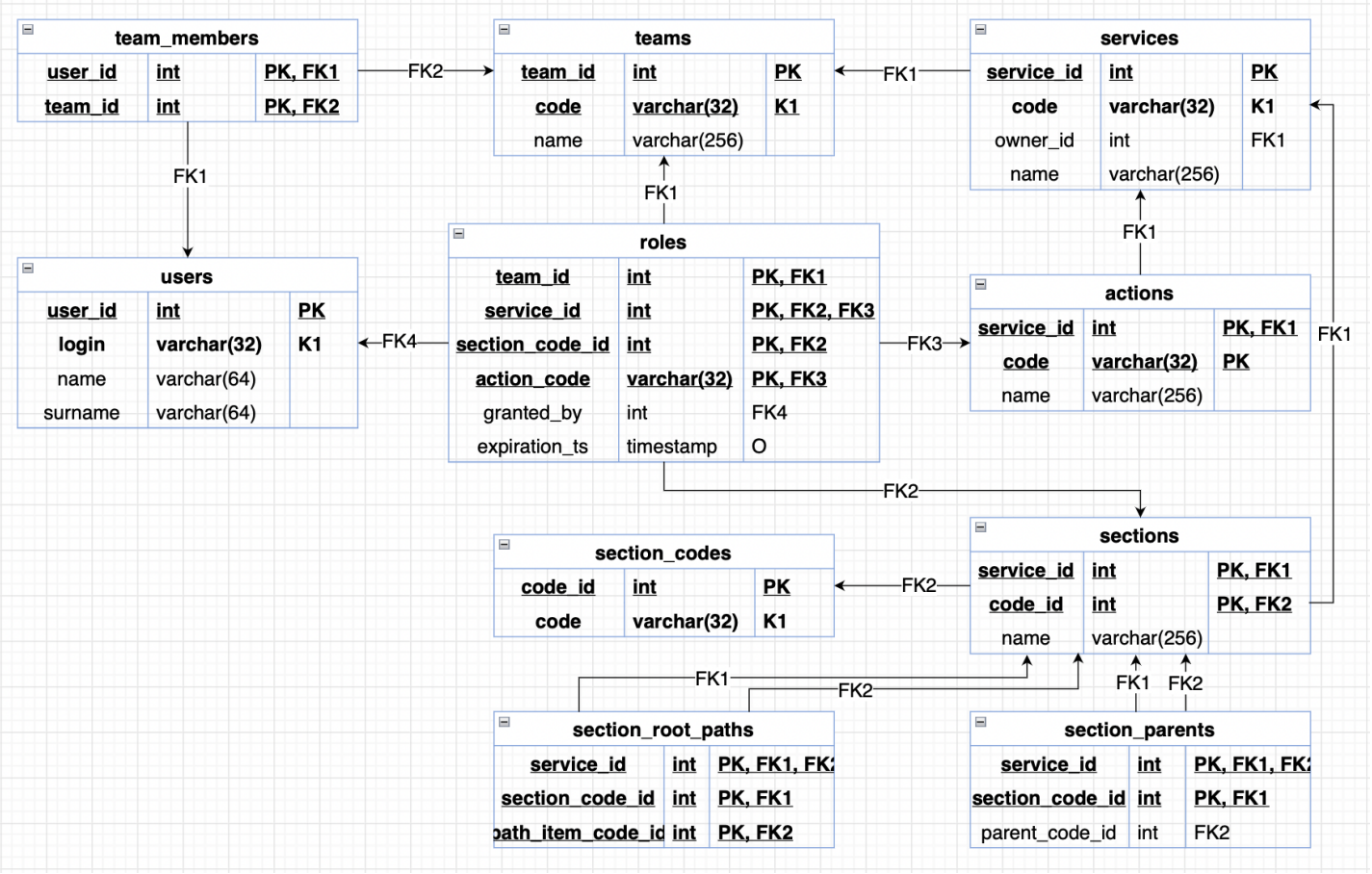
Модель сущность-связь



Тут нет информации о section_parents и section_root_paths, так как я посчитал, что это скорее детали физической модели (то, как именно мы храним связь “parent”).

Также отдельно хочу отметить разницу между actions и sections. Обе они - зависимые сущности от service, но одна идентифицируется напряму по коду, а вторая - по выделенному отдельно code_id. Разница эта обусловлена тем, что sections - сущность иерархическая, поэтому ее primary key (который будет (service_id, code_id)) мы будем хранить очень часто (в случе, если пользователи соберут бамбук из n секций, мы получим O(n^2) записей в section_root_paths) => не хотелось хранить каждый раз varchar(32) и я решил вынести section_code отдельно, сохранив при этом свойство уникальности кода секции в рамках конкретного сервиса.

Физическая модель



Тут хочу отметить появление таблиц section_parents и section_root_paths. Можно было сделать parent_code_id nullable полем в sections, но я посчитал нулы тут плохой идеей, поэтому section_parents - отдельная таблица.

Таблица section_root_paths будет обновляться триггерами и не подразумевает прямых апдейтов. Валидация на отсутствие циклов в section_parents также будет выполняться триггерами. Изначально я планировал делать section_root_paths как materialized view, но после ознакомления с тем, как именно postgresql (не) поддерживает такие представления, я решил, что гораздо оптимальнее будет пересчитывать предков триггерами (получилось гораздо приятнее, чем я думал).

При построении физической модели использовалось следующее отображение доменов в типы:

Домен	Тип
`id`	`int`
`login`	`varchar(32)`
`user_name`	`varchar(64)`
`user_surname`	`varchar(64)`
`*_code`	`varchar(32)`
`*_name`	`varchar(256)`
`utc_date_time`	`timestamp`

Определения таблиц

Для реализации проекта использовалась СУБД postgres 14. Определения таблиц и их индексов приведено в файле `ddl.sql`.

Тестовые данные

Скрипт для добавления тестовых данных приведен в файле `examples.sql`.

Запросы на получение данных

В рамках проекта были реализованы следующие запросы:

- user_has_role_exact — выводит, есть ли у пользователя ровно такая роль, полученная не транзитивно
- user_has_role_at_least — выводит, есть ли у пользователя такая роль, возможно, полученная транзитивно
- users_can_grant_role — выводит пользователей, которые могут выдать определенную роль
- roles_expiring_tomorrow — выводит роли, истекающие в ближайшие 24 часа
- Most_granting_users — выводит пользователей, которые выдали больше всего ролей

- minimal_required_section — выводит по двум заданным секциям такую, на которую можно выдать роль, которая порастет по транзитивности в обе заданные секции (на самом деле, просто lca)
- useless_actions — показывает действия, на которые не было выдано ни одной роли
- service_granted_actions_count — для каждого сервиса показывает количество действий, на которые выдана хотя бы одна роль.
- average_user_roles_by_team — для каждой команды выводит среднее количество ролей у этого пользователя (может быть не целым, тк юзер может быть привязан к разным командам)

Для реализации запросов были созданы вспомогательные представления:

- full_section_path — представление, используя которое, можно получить путь от секции до ее корня по коду сервиса и секции
- team_service_actions — представление, содержащее все активные роли, выданные на все команды, позволяет получать роли по кодам соответствующих сущностей

Запросы на получение данных и вспомогательные представления приведены в файле `selects.sql`.

Запросы на изменение данных

В рамках проекта были реализованы следующие запросы:

- fn__grant_role — процедура выдачи роли с проверкой прав доступа
- fn__extend_role — процедура изменения даты протухания роли с проверкой прав доступа
- fn__revoke_role — процедура отзыва роли с проверкой прав доступа
- fn__change_service_owner — процедура изменения команды-владельца сервиса (а, следовательно, и пользователей, выдающих на этот сервис роли), с проверкой прав доступа

Также в файле ddl.sql есть две интеллектуальные хранимые процедуры:

- fn__section_root_paths__add_edge — процедура добавления ребра в таблицу section_root_paths, используется в триггерах
- fn__section_root_paths__delete_edge — процедура удаления ребра из таблицы, используется в триггерах

Запросы на изменение данных, хранимые процедуры и триггеры приведены в файле `updates.sql`. Запросы вида “insert into users(...)” я решил не приводить ввиду их абсолютной неинтересности (там в хранимках есть инсерты повеселее). Все хранимые процедуры корректно работают с уровнем изоляции read committed.