1. Идея приложения и схема OLTP

Идея приложения

Пожалуй, начнем с самой идеи приложения, чтобы было понятно о чем идет речь.

У нас есть приложение, хранящее “mediafiles” - абстрактное понятие, объединяющее понятие фото и видео. Поскольку базы данных не очень хорошее решение для хранения гигабайтов бинарных данных (хотя почему бы и нет?), но в данном случае мы используем S3-совместимое хранилище с разными регионами и серверами, храня лишь ссылку на сам файл. Также у медиафайла есть галочка “is_favorite”, означающая что это “любимый” файл пользователя и “trashed_datetime” - время отправки файла в корзину.

У медиафайлов есть пользователи, ими владеющие. У каждого файла может быть только один “автор”, система взаимодействия между пользователями не предусмотрена, т.е. никаких общих альбомов или шаринга - у нас тут MVP пока что :)

У каждого из пользователей есть какие-то предпочтения. Я фанат давать пользователю максимальное количество возможных настроек, поэтому как только я пойму, что должно быть в реальном приложении в этом разделе - таблица вырастет значительно. В связи с этим два жалких поля вынесены в отдельную таблицу user_preferences.

Категоризация... Пользователю нужно дать возможность как-то организовывать свою библиотеку файлов, иначе она быстро превратиться в слабо управляемый хаос. В связи с этим у нас есть “albums” и “tags”. Альбом - условная “папка” для фото, без вложенности или дополнительной информации. У неё есть имя и некоторая сервисная информация, вроде времени создания, которая нужна для сортировки папок в интерфейсе приложения. Теги - ключевые слова, используемые в основном в качестве фильтров для поиска. Разница между обычным поиском по названиям и подходом с тегами в том, что запрос:

```
cats Ostin
```

при классическом подходе выведет кота по имени Остин, но вот такой вариант:

```
Ostin cats
```

работать не будет, вы получите ноль результатов, потому что нету таких котов, которые бы “произошли” от Остина. (вставьте сюда шутку про троянского кота)

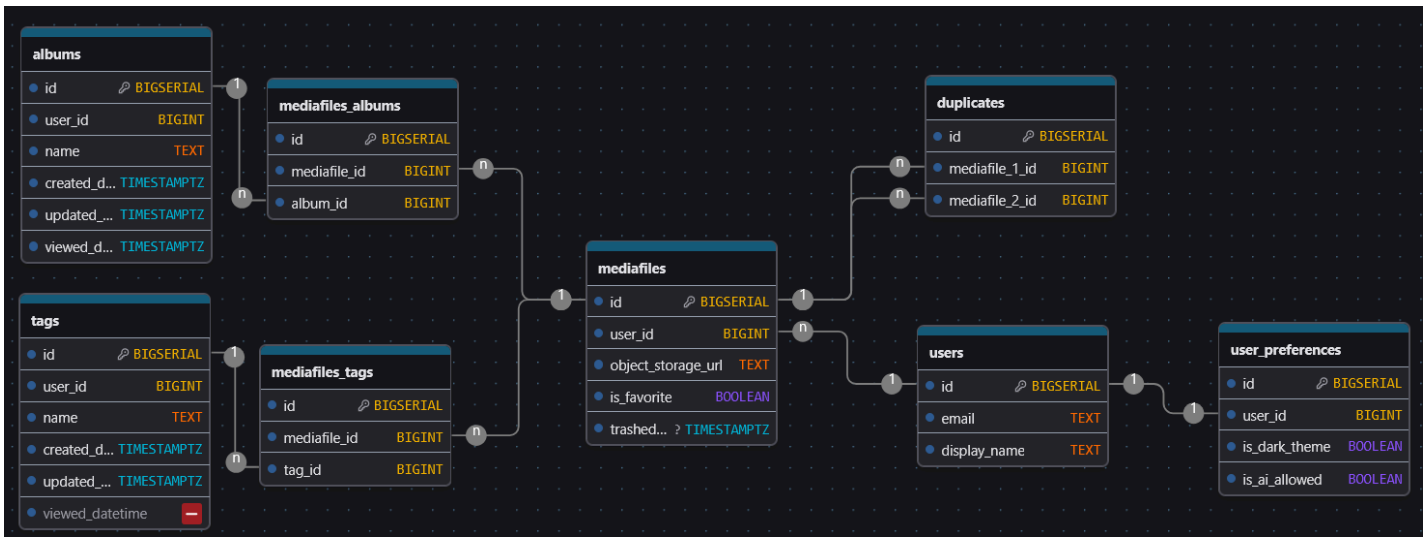
Теги решают эту проблему. Зная, какие теги есть у него в его собственной галерее, пользователь может за считанные секунды собирать сложно-фильтрованные запросы, на основе заранее сделанных “фильтров”, коими выступают названия тегов. Самое крутое в этом техническом решении - максимальная гибкость. Пользователь сам решит, что ему важнее вставлять в эти названия и какой структуры придерживаться. Но это в идеальном мире. А в нашем...

Это требует некоторой предварительной работы от пользователя, которую не всем хочется делать. Именно поэтому мы имеем галочку на разрешение использовать AI в настройках - пользователь может разрешить нейронной сети создать ключевые слова для каждой картинки.

И, наконец, дубликаты. Это файлы, помеченные как дубликаты со стороны приложения. Такое может произойти в случае, если пользователь сохраняет картинки из интернета/соц. сетей и в какой-то момент скачает одну и ту же картинку с разных источников. Механизм определения дубликатов не так важен для этого проекта, поэтому оставляю его за кулисами.

Схема OLTP

Таким образом, с учетом всех данных, моделируем подобную БД для работы приложения:



P.S. Схему можно посмотреть в интерактивном режиме, загрузив её сюда:

https://www.drawdb.app/editor

Для этого перейдите по ссылке выше, выберите в меню Файл→Импорт→JSON и добавьте туда следующий файл:

> Photos_app_2025-06-09T13_19_12.178Z.json

Также схема доступна в виде SQL-файла для создания базы в репозитории проекта:

```
./db_creation_scripts/oltp/oltp_create.sql
```

С моей точки зрения, схема в совокупности с файлом создания базы и моим объяснением модели данных достаточно неплохо описывают происходящее в базе. Из дополнительных пояснений к ключам/таблицам/связям/constraints могу добавить разве что информацию касательно “unique” и “check” в некоторых таблицах.

Причины их появления - две:

- Так проще и быстрее заполнять некоторые таблицы. К примеру, в приложении по умолчанию отключено ИИ (уважаем конфиденциальность пользователей) - а значит, он может быть false и в базе данных.
- В некоторых случаях использование check позволяет существенно упростить логику etl, при этом не накладывая дополнительных логических проблем на изначальную схему. К примеру: в таблице дубликатов стоит при добавлении проверять, что первый из двух файлов всегда имеет айди меньше, чем второй. Эта простая и удобная проверка позволяет избежать проблем не только с ETL, но и при заполнении таблиц - чтобы не плодить дубликаты записей о дубликатах :)

Мне эти причины кажутся достаточно весомыми для использования таких ограничений. Кстати всё то же самое касается и базы OLAP тоже.