

## 6. Управление транзакциями: механизмы SQL и практика в PostgreSQL

### Практическая часть: управление транзакциями в PostgreSQL на примере Python и psycopg2

В PostgreSQL, как и в других реляционных СУБД, транзакции управляются через три базовые SQL-команды:

- **BEGIN** (или **START TRANSACTION**) — открывает новую транзакцию. Все изменения, сделанные после этой команды, считаются частью одной логической операции.
- **COMMIT** — завершает транзакцию, делая изменения постоянными в базе данных.
- **ROLLBACK** — отменяет все изменения, произведённые после **BEGIN**, в случае ошибки или отмены операции.

Эти команды особенно важны в сценариях, где ошибки могут возникать в произвольный момент выполнения — например, при валидации данных, сетевых сбоях или нарушении ограничений (например, уникальности или **CHECK**).

Транзакции позволяют гарантировать, что либо все действия выполняются успешно, либо не сохраняется ничего, тем самым обеспечивая **атомарность** и **согласованность** (ACID).

#### Пример: использование транзакций в Python-коде

Для демонстрации мы используем библиотеку `psycopg2`, которая предоставляет интерфейс к PostgreSQL из Python. В рамках практической части нас интересует **только** функция `seed_accounts_transactional`, так как именно она демонстрирует транзакционную логику. Функции `clear_db()` и `print_table()` необходимы лишь для удобства тестирования и вывода результата, но к теме транзакций непосредственно не относятся.

Эти команды особенно важны в сценариях, где ошибки могут возникнуть в любой момент — при валидации данных, нарушении ограничений или, например, в случае сетевых сбоев. Они позволяют контролировать, в каком именно состоянии данные попадут в базу или будут отменены.

Теперь перейдём к коду.

Сначала обратим внимание на начало функции. Здесь используется строка `conn.set_session(autocommit=False)`. Это означает, что автоматическая фиксация транзакций отключена, и теперь вся работа с базой будет происходить внутри явной транзакции. Это эквивалентно выполнению команды `BEGIN` в SQL.

Далее начинается основной цикл, в котором происходит попытка вставки новых записей в таблицу `accounts`. Для каждой пары имя-владелец и баланс сначала проверяется, существует ли уже аккаунт с таким именем. Если пользователь найден, вставка пропускается, чтобы избежать ошибки из-за ограничения `UNIQUE`.

Если имя уникально, выполняется вставка с указанием владельца и начального баланса. Однако важно понимать, что это может не сработать, если, например, баланс отрицательный — в таблице есть ограничение `CHECK (balance >= 0)`. В этом случае PostgreSQL вызовет ошибку.

Теперь главное: вся логика вставки обернута в блок `try`. Это значит, что если при добавлении хотя бы одной строки произойдёт ошибка — например, попытка вставить дубликат или отрицательный баланс — мы попадаем в `except`, где выполняется `conn.rollback()`. Это команда `ROLLBACK`, которая отменяет **все изменения**, сделанные после начала транзакции. Даже если до этого пять других строк вставились корректно, они тоже будут отменены.

Если же ошибок не возникло, то в конце вызывается `conn.commit()` — транзакция фиксируется, и все изменения становятся постоянными.

Таким образом, функция реализует полный цикл работы с транзакцией: начало (BEGIN через `autocommit=False`), выполнение операций, завершение через COMMIT или отмена через ROLLBACK при ошибке. Это позволяет гарантировать, что данные либо сохраняются целиком, либо не сохраняются вообще — как и требует принцип атомарности в ACID.

### **SAVEPOINT и частичный откат**

Теперь перейдём ко второй версии функции `seed_accounts_transactional`, в которой используется новый инструмент — **SAVEPOINT**.

Ранее мы уже включали управление транзакцией вручную, отключая `autocommit`. Это позволяло начать транзакцию (аналог команды BEGIN) и в случае ошибки откатить все изменения с помощью ROLLBACK. Однако в таком подходе даже одна ошибка приводит к полной потере всего, что уже было сделано в рамках транзакции.

Чтобы решить эту проблему, мы добавляем механизм **SAVEPOINT**.

**SAVEPOINT** — это точка сохранения внутри текущей транзакции. Она создаёт "промежуточную метку", к которой можно откатиться в случае локальной ошибки, не прерывая всю транзакцию. Это как «контрольная точка»: если что-то пошло не так, мы возвращаемся к этой точке и продолжаем выполнение с неё.

Это позволяет отменить **только неудачную попытку**, при этом все предыдущие успешные вставки сохраняются в рамках текущей транзакции. В конце, когда цикл завершается, вызывается `conn.commit()` — и все добавленные корректно записи фиксируются.

Такой подход даёт гораздо большую гибкость. Давайте сравним поведение с предыдущей версией:

- В первом варианте (без SAVEPOINT) — одна ошибка приводит к полному откату: база остаётся пустой, даже если большинство записей были валидны.
- Во втором варианте (с SAVEPOINT) — при ошибке мы просто «откатываемся на шаг назад» и продолжаем. В результате в таблице оказываются все те аккаунты, вставка которых прошла успешно.

Таким образом, SAVEPOINT — это механизм для **вложенного управления ошибками** в транзакциях. Он позволяет изолировать и локально обрабатывать сбои, не обрывая всю транзакцию. Это особенно важно, если нам нужно загрузить большую партию данных, среди которых могут встретиться ошибки, но мы не хотим терять всё из-за одного сбоя.

Теперь рассмотрим следующий уровень работы с транзакциями — параллельное выполнение и уровни изоляции. Для этого в коде появляются две новые функции: `transfer` и `simulate_lost_update`.

### **Функция `simulate_lost_update`: моделирование гонки данных**

Теперь мы переходим к следующему этапу — **параллельному выполнению транзакций** и работе с **уровнями изоляции**. Для этого в нашем коде появляются две новые функции: `transfer` и `simulate_lost_update`.

Сначала — про `transfer`.

Эта функция отвечает за перевод средств между двумя аккаунтами. Она работает внутри одной транзакции и поддерживает выбор уровня изоляции. Основной акцент здесь сделан на том, что строка отправителя блокируется до конца транзакции. Это реализовано

с помощью механизма блокировки — чтобы никакая другая параллельная транзакция не могла изменить эту строку, пока текущая операция не будет завершена.

Далее, в рамках одной логической операции, сначала считывается баланс отправителя, затем проверяется его достаточность. Если всё в порядке — списываются средства у одного и зачисляются другому. Плюс, результат переводов логируется в отдельную таблицу транзакций. Если же где-то происходит сбой — транзакция откатывается целиком, чтобы сохранить целостность данных.

Теперь к самой интригующей части — `simulate_lost_update`. Здесь мы специально моделируем **потерянное обновление** — одну из самых известных и опасных проблем при параллельных транзакциях. Сценарий предельно простой: запускаются два потока, и каждый из них одновременно пытается списать средства с одного и того же счёта, но перевести их разным получателям.

Мы создаём идеальные условия для возникновения ошибки:

- Параллельное выполнение в двух потоках
- Отсутствие искусственной задержки между чтением и записью
- Использование даже базового уровня изоляции — `READ COMMITTED`

Однако результат оказался неожиданным.

**PostgreSQL не допустил потери данных.** Даже при минимально допустимом уровне изоляции все переводы были обработаны корректно: итоговый баланс сходился, транзакции не перезаписывали друг друга, а в таблице появлялись все ожидаемые записи.

Почему так произошло? Дело в том, что **механизм блокировок в PostgreSQL работает глубже, чем того требует стандарт SQL**. Благодаря использованию MVCC и команд вроде `SELECT ... FOR UPDATE`, строки, к которым обращается каждая

транзакция, блокируются автоматически. Это означает, что один поток будет ждать завершения другого, прежде чем продолжить.

В результате, даже в условиях конкурентного доступа, PostgreSQL обеспечивает корректность выполнения и защищает нас от потери обновлений — при условии, что мы правильно используем блокировки.