**Требования ACID** — набор требований, которые обеспечивают сохранность ваших данных. Что особенно важно для финансовых операций. Мы же не хотим остаться без денег из-за разрыва соединения или ошибки в ПО, не так ли?

Давайте пройдемся по каждой букве ACID и посмотрим на примерах, чем архив лучше 10 разных файлов. И чем транзакция лучше 10 отдельных запросов.

**Atomicity — Атомарность**

Атомарность гарантирует, что каждая транзакция будет выполнена полностью или не будет выполнена совсем. Не допускаются промежуточные состояния.

Друг познается в беде, а база данных — в работе с ошибками. О, если бы всё всегда было хорошо и без ошибок! Тогда бы никакие ACID были бы не нужны. Но как только возникает ошибка, атомарность становится очень важна.

Допустим, вы решили отправить маме деньги. Когда вы делаете перевод внутри банка, что происходит:



И допустим, что у нас 2 отдельных запроса. А теперь посмотрим, что будет при возникновении ошибок:

1. У вас на балансе нет нужной суммы — система вывела сообщение об ошибке, но катастрофы не произошло, атомарность тут не нужна.
2. У мамы заблокирована карточка, истек срок годности — деньги ей не поступили. Запрос отменен. Но минуточку... У вас то они уже списались!

Ошибка на первом этапе никаких проблем в себе не таит. А вот ошибка на втором... Приводит к потере денег, что явно недопустимо.

Если мы отправляем отдельные запросы, система не может связать их между собой. Запрос упал с ошибкой? Система его отменяет. Но только его, ведь она не знает о том, что запрос «у меня деньги спиши» связан с упавшим «сюда положи»!

Транзакция же позволяет сгруппировать запросы, то есть фактически показывает базе на взаимосвязи между ними. База сама о связях ничего не знает! Это знаете только вы =)

И если падает запрос внутри транзакции, база откатывает всю транзакцию. И приходит в состояние «как было до начала транзакции». Даже если там внутри было 10 запросов, вы можете спать спокойно — сломался один, откатятся все.

**Consistency — Согласованность**

Транзакция, достигающая своего нормального завершения (EOT — end of transaction, завершение транзакции) и, тем самым, фиксирующая свои результаты, сохраняет согласованность базы данных. Другими словами, каждая успешная транзакция по определению фиксирует только допустимые результаты

Это свойство вытекает из предыдущего. Благодаря тому, что транзакция не допускает промежуточных результатов, база остается консистентной. Есть такое определение транзакции: «**Упорядоченное множество операций**, **переводящих базу данных из одного согласованного состояния в другое**». То есть до выполнения операции и после база остается консистентной (в переводе на русский — согласованной).

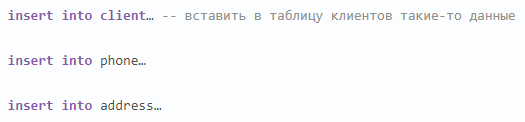
Например, пользователь в системе заполняет карточку:

* ФИО
* Дата рождения
* ИНН
* Телефон — отдельно код страны, города и номер
* Адрес — тоже разбит на несколько полей

В базе данных у нас есть несколько таблиц:

* client
* phone
* address

Так что когда пользователь заполнил форму и нажал «сохранить», система отправляет в базу данных 3 запроса:



Можно отправить 3 разных запроса, но лучше сделать одну транзакцию, внутри которой будут эти 3 запроса.

Атомарность гарантирует, что не получится такого, что адрес с телефоном сохранились, а сам клиент — нет. Это сделало бы базу неконсистентной, ведь у нас бы появились атрибуты, «висящие в воздухе», никому не принадлежащие. Что, в свою очередь, приведет к ошибкам в системе.

За консистентностью должен следить разработчик. Ведь это вопрос скорее бизнес-логики, чем технологий. Те же атрибуты, «висящие в воздухе» — это разработчик знает, что:

* если есть телефон в таблице phone
* он должен ссылаться на таблицу client

База об этом не знает ничего, если ей не рассказать. И она легко пропустит запрос «добавь в базу телефон без ссылки на клиента», если сам по себе запрос корректный, а разработчик не повесил на таблицу foreign key.

Можно повесить на таблицу **constraint**. Например, «**баланс строго положительный**». Тогда сценарий с ошибкой будет выглядеть так:

1. Пользователь пытается перевести другу 100р, хотя у него самого 10
2. Система отправляет в базу запрос — «обнови баланс карты, теперь там X – 100».
3. База пытается выполнить запрос, но ой! Нарушен constraint, в итоге операции баланс стал отрицательным, эту ошибку она и возвращает.
4. Система обрабатывает ошибку и выводит ее пользователю в читаемом виде.

К сожалению, нет единого механизма рассказать базе о том, какое состояние считается согласованным. Разработчик может использовать

* foreign ключи
* какие-то констрейнты — это БД проверит.

Но что с одного счета списалось, а на другой пришло — это БД уже не проверит. Это бизнес-логика.

Разработчик пишет код, пошагово переводящий БД в нужное согласованное состояние и, если где-то посередине возникает ошибка или нежданчик, откатывает всю транзакцию. То есть можно после каждого шага делать запрос, проверяя какое-то поле:

— Эй, баланс, ты ведь положительный остался?

— Ку-ку, тебе деньги пришли?

Если вдруг проверка не прошла, то кидаем ошибку и делаем откат.

**Isolation — Изолированность**

**Во время выполнения транзакции параллельные транзакции не должны оказывать влияния на её результат.**

Если у нас система строго для одного человека, проблем не будет. А если пользователей несколько? Тогда транзакции запускают в параллель — для ускорения работы системы. А иначе представьте себе, что вы делаете заказ в интернет-магазине и система вам говорит: «Вы в очереди, перед вами еще 100 человек хотят заказ оформить, подождите». Бред же? Бред!

Вот и приходится распараллеливать запросы. Но к каким эффектам может привести параллельная работа двух транзакций?

**1 эффект: "Потерянная запись"**



* 1. Есть некий счет А, на котором лежит 500 у.е.
  2. Кассир 1 (К1 на рисунке) списал с него 300 у.е. Обозначим его действия рыжими стрелками. Списал 300, на выходе получает 200 = 500 - 300.
  3. Кассир 2 (К2) тоже решил обратиться к этому же счету, и записал туда 300 у.е., пока К1 еще не успел закрыть свою транзакцию. Так как первая транзакция не закрыта, сумма на счете до сих пор 500, получаем 500 + 300 = 800.

Итог — мы "потеряли запись" первого кассира, ведь на выходе у нас А = 800, хотя должно быть 500. "Кто последний вписал результат - того и тапки". Получается так.

**2 эффект: "Грязное чтение"**



1. Есть некий счет А, на котором лежит 500 у.е.
2. Кассир 1 списал с него 300 у.е. Обозначим его действия рыжими стрелками. Списал 300. Потом передумал и сделал откат - на выходе остались те же 500 у.е.
3. Кассиру 2 (К2) понадобилась информация по этому счету и он ее считал до того, как К1 закрыл свою транзакцию.

Итог — второй кассир считал неверную сумму, построил неверный отчет/отказал в визе платежеспособному гражданину и т.д.

**3 эффект: "Повторимое чтение"**



1. Есть некие данные.
2. Кассир 1 строит отчет. Операции идут последовательно для каждой колонки. Система считала данные, записала в первую колонку (например, взяв минимум от них). Обозначим получение данных зеленым цветом, а изменение - рыжим.
3. Кассир 2 влез в эту таблицу данных и изменил некоторые счета в ней.
4. У кассира 1 продолжается построение отчета. И во вторую колонку система считывает уже новые данные.

Итог - отчет построен на основании разных данных.

**4 эффект: "Фантомы"**



1. Есть некие данные.
2. Кассир 1 строит отчет. Операции идут последовательно для каждой колонки. Система считала данные, записала в первую колонку (например, взяв минимум от них). Обозначим получение данных зеленым цветом, а изменение - рыжим.
3. Кассир 2 влез в эту таблицу данных и добавил новые счета/удалил некоторые старые.
4. У кассира 1 продолжается построение отчета. И во вторую колонку система считывает уже новые данные.

Итог — отчет построен на основании разных данных.

Разница между 3-им и 4-ым эффектами в том, что в одном случае данные изменяются, а во втором — добавляются/удаляются. То есть меняется ещё и их количество.

**Как бороться**

Как бороться с этими проблемами? Нужно изолировать транзакцию. Способов есть несколько, но основные

* блокировки
* версии

**Блокировки** — это когда мы блокируем данные в базе. Можно заблокировать одну строку в таблице, а можно всю таблицу. Можно заблокировать данные на редактирование, а можно и на чтение тоже.

**Версии** — это когда внутри базы при каждом обновлении создается новая версия данных и сохраняется старая. Версионирование скрыто от разработчика, то есть мы не видим в базе никаких номеров версий и данных по ним. Просто пока транзакция, обновляющая запись, не покомитит свое изменение, остальные потребители читают старую версию записи и не блокируются.

**Durability — Надёжность**

Если пользователь получил подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им изменения не будут отменены из-за какого-либо сбоя. Обесточилась система, произошел сбой в оборудовании? На выполненную транзакцию это не повлияет.