1. Что такое ACID?

https://habr.com/ru/post/317884/

Транзакции — это группа операций на чтение/запись, выполняющихся только если все операции из группы успешно выполнены.

По сути транзакции характеризуются следующими четырьмя свойствами (также известными как ACID):

1. Атомарность
2. Консистентность
3. Изоляция
4. Долговечность

### Атомарность

Атомарность позволяет объединить единые и независимые операции в «единицу работы», которая работает по принципу «все-или-ничего», успешно выполняющаяся только если все содержащиеся операции успешно выполнятся.

Транзакция может инкапсулировать изменение состояний. Транзакция должна всегда оставлять систему в консистентном состоянии, независимо от того сколько параллельных транзакций выполняются в любой промежуток времени.

### Консистентность (согласованность)

Консистентность означает что все требования уникальности были соблюдены для каждой совершенной транзакции. Это подразумевает, что требования по всем ключам (primary и foreign key), типам данных, триггерам успешно пройдены и не было найдено нарушений требования уникальности.

### Изоляция

Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат. Изоляция дает нам преимущество сокрытия нефинальных изменений состояния от внешнего мира, и так же неуспешные транзакции не должны никогда порушить состояние системы. Изоляция достигается через управление параллельным выполнением операций используя пессимистический или оптимистический механизм блокировки.

### Долговечность

Успешная транзакция должна всегда изменять состояние системы и прежде чем закончить ее изменения состояния сохраняются в лог транзакций. Если Ваша система внезапно выключится или возникнет перебой с электричеством, тогда все незавершенные транзакции можно воспроизвести.

Для систем сообщений, как JMS, транзакции не являются обязательными. Именно по этой причине в спецификации есть режимы подтверждения вне транзакций.

Операции с файловой системой обычно не контролируются, но если бизнес-требования требуют транзакций для операций с файлами, Вы можете использовать инструмент, такой как XADisk.

В то время как для файловых систем и систем передачи сообщений использование транзакций опционально, то для БД контроль за транзакциями обязателен.

https://ru.bmstu.wiki/ACID_(Atomicity,_Consistency,_Isolation,_Durability)

2. Уровни изолированности транзакций

https://akorsa.ru/2016/08/rukovodstvo-dlya-nachinayushhih-acid-i-tranzaktsii-bd/

https://ru.wikipedia.org/wiki/Уровень_изолированности_транзакций#Потерянное_обновление

**Уровень изолированности транзакций** — условное значение, определяющее, в какой мере в результате выполнения логически параллельных транзакций в СУБД допускается получение несогласованных данных. Шкала уровней изолированности транзакций содержит ряд значений,

проранжированных от наинизшего до наивысшего; более высокий уровень изолированности соответствует лучшей согласованности данных, но его использование может снижать количество физически параллельно выполняемых транзакций.

Стандарты SQL определяют 4 уровня изолированности:

- READ_UNCOMMITTED
- READ_COMMITTED
- REPEATABLE_READ
- SERIALIZABLE

Все уровни, кроме SERIALIZABLE уровня подвержены аномалии данных (феномен), которые могут произойти согласно следующей схеме:

| Уровень изоляции | «Грязное» чтение | Неповторяемое чтение | Фантомное чтение | Аномалия сериализации |
|---|---|---|---|---|
| Read uncommited (Чтение незафиксированных данных) | Допускается, но не в PG | Возможно | Возможно | Возможно |
| Read committed (Чтение зафиксированных данных) | Невозможно | Возможно | Возможно | Возможно |
| Repeatable read (Повторяемое чтение) | Невозможно | Невозможно | Допускается, но не в PG | Возможно |
| Serializable (Сериализуемость) | Невозможно | Невозможно | Невозможно | Невозможно |

3. Что такое «Потерянное обновление (Lost Update)»?

Ситуация, когда при одновременном изменении одного блока данных разными транзакциями одно из изменений теряется.

Предположим, имеются две транзакции, выполняемые одновременно:

| Транзакция 1 | Транзакция 2 |
|---|---|
| `UPDATE tbl1 SET f2=f2+20 WHERE f1=1;` | `UPDATE tbl1 SET f2=f2+25 WHERE f1=1;` |

В обеих транзакциях изменяется значение поля f2, по их завершении значение поля должно быть увеличено на 45. В действительности может возникнуть следующая последовательность действий:

1. Обе транзакции одновременно читают текущее состояние поля. Точная физическая одновременность здесь не обязательна, достаточно, чтобы вторая по порядку операция чтения выполнилась до того, как другая транзакция запишет свой результат.
2. Обе транзакции вычисляют новое значение поля, прибавляя, соответственно, 20 и 25 к ранее прочитанному значению.
3. Транзакции пытаются записать результат вычислений обратно в поле f2. Поскольку физически одновременно две записи выполнить невозможно, в реальности одна из операций записи будет выполнена раньше, другая позже. При этом вторая операция записи перезапишет результат первой.

В результате значение поля f2 по завершении обеих транзакций может увеличиться не на 45, а на 20 или 25, то есть одна из изменяющих данные транзакций «пропадёт».

4. Что такое «Грязное чтение (Dirty Read)»?

Чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится).

Предположим, имеются две транзакции, открытые различными приложениями, в которых выполнены следующие SQL-операторы:

| Транзакция 1 | Транзакция 2 |
|---|---|
| `UPDATE tbl1 SET f2=f2+1 WHERE f1=1;` | |
| | `SELECT f2 FROM tbl1 WHERE f1=1;` |
| `ROLLBACK WORK;` | |

В транзакции 1 изменяется значение поля f2, а затем в транзакции 2 выбирается значение этого поля. После этого происходит откат транзакции 1. В результате значение, полученное второй транзакцией, будет отличаться от значения, хранимого в базе данных.

5. Что такое «Неповторяющееся чтение (Non-Repeatable Read)»?

Ситуация, когда при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными.

Предположим, имеются две транзакции, открытые различными приложениями, в которых выполнены следующие SQL-операторы:

| Транзакция 1 | Транзакция 2 |
|---|---|
| | `SELECT f2 FROM tbl1 WHERE f1=1;` |
| `UPDATE tbl1 SET f2=f2+1 WHERE f1=1;` | |
| `COMMIT;` | |

| | |
|---|---|
| | `SELECT f2 FROM tbl1 WHERE f1=1;` |

В транзакции 2 выбирается значение поля f2, затем в транзакции 1 изменяется значение поля f2. При повторной попытке выбора значения из поля f2 в транзакции 2 будет получен другой результат. Эта ситуация особенно неприемлема, когда данные считываются с целью их частичного изменения и обратной записи в базу данных.

6. Что такое «Чтение фантомов (Phantom Reads)»?

Ситуация, когда при повторном чтении в рамках одной транзакции одна и та же выборка дает разные множества строк.

Предположим, имеется две транзакции, открытые различными приложениями, в которых выполнены следующие SQL-операторы:

| Транзакция 1 | Транзакция 2 |
|---|---|
| | `SELECT SUM(f2) FROM tbl1;` |
| `INSERT INTO tbl1 (f1,f2) VALUES (15,20);` | |
| `COMMIT;` | |
| | `SELECT SUM(f2) FROM tbl1;` |

В транзакции 2 выполняется SQL-оператор, использующий все значения поля f2. Затем в транзакции 1 выполняется вставка новой строки, приводящая к тому, что повторное выполнение SQL-оператора в транзакции 2 выдаст другой результат. Такая ситуация называется чтением фантома (фантомным чтением). От неповторяющегося чтения оно отличается тем, что результат повторного обращения к данным изменился не из-за изменения/удаления самих этих данных, а из-за появления новых (фантомных) данных.

7. Что такое «Аномалии сериализации»?

Например, рассмотрим таблицу `mytab`, изначально содержащую:

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

Предположим, что сериализуемая транзакция A вычисляет:

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

а затем вставляет результат (30) в поле `value` в новую строку со значением `class = 2`. В это же время сериализуемая транзакция B вычисляет:

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

получает результат 300 и вставляет его в новую строку со значением `class = 1`. Затем обе транзакции пытаются зафиксироваться. Если бы одна из этих транзакций работала в режиме Repeatable Read, зафиксироваться могли бы обе; но так как полученный результат не соответствовал бы последовательному порядку, в режиме Serializable будет зафиксирована только одна транзакция, а вторая закончится откатом с сообщением:

```
ОШИБКА: не удалось сериализовать доступ из-за зависимостей чтения/записи между
  транзакциями
```

Это объясняется тем, что при выполнении A перед B транзакция B вычислила бы сумму 330, а не 300, а при выполнении в обратном порядке A вычислила бы другую сумму.

Рассчитывая, что сериализуемые транзакции предотвратят аномалии, важно понимать, что любые данные, полученные из постоянной таблицы пользователя, не должны считаться действительными, пока транзакция, прочитавшая их, не будет успешно зафиксирована. Это верно даже для транзакций, не модифицирующих данные, за исключением случая, когда данные считываются в *откладываемой* транзакции такого типа. В этом случае данные могут считаться действительными, так как такая транзакция ждёт, пока не сможет получить снимок, гарантированно предотвращающий подобные проблемы. Во всех остальных случаях приложения не должны полагаться на результаты чтения данных в транзакции, которая не была зафиксирована; в случае ошибки и отката приложения должны повторять транзакцию, пока она не будет завершена успешно.

8. Что такое MVCC? Для чего нужно и как работает?

https://pgcookbook.ru/articles/mvcc_and_vacuum.html

https://habr.com/ru/post/208400/

**MVCC** – одна из ключевых технологий доступа к данным, которая используется в PostgreSQL. Она позволяет осуществлять параллельное чтение и изменение записей (tuples) одних и тех же таблиц без блокировки этих таблиц. Чтобы иметь такую возможность, данные из таблицы сразу не удаляются, а лишь помечаются как удаленные. Изменение записей осуществляется путем маркировки этих записей как удаленных, и созданием новых записей с измененными полями. Таким образом, история изменений одной записи сохраняется в базе данных и доступна для чтения другими транзакциями. Этот способ хранения записей позволяет параллельным процессам иметь неблокирующий доступ к записям, которые были удалены или изменены в параллельных незакрытых транзакциях. Техника, используемая в этом подходе, относительно простая. У каждой записи в таблицы есть системные скрытые поля xmin, xmax.

- **xmin** – хранит номер транзакции, в которой запись была создана.
- **xmax** – хранит номер транзакции, в которой запись была удалена или изменена.

С помощью этих полей и осуществляется фильтрация записей (есть еще поля cmax/cmin, на них я не буду останавливаться). Перед началом выборки данных PostgreSQL сохраняет *снапшот* текущего состояния БД. На основании данных снапшота, полей xmin, xmax осуществляется фильтрация записей.

**В очень упрощенном виде работу MVCC можно представить таким образом**

Если к таблице *table* делается запрос

```
SELECT * FROM table WHERE ваши условия
```

PostgreSQL обрабатывает запрос и добавляет в него доп. условия

```
SELECT * FROM table WHERE ваши условия AND is_tuple_visible(xmin, xmax, snapshot)
```

где is_tuple_visible – функция, которая по значениям полей записи xmin и xmax и сохраненного снапшота определяет, видна ли эта запись для текущего запроса

У данной технологии есть одна неприятная особенность. Суть её заключается в том, что при изменении записи, в таблицу добавляется новая запись, а старая копия помечается как удаленная, что ведет к росту размера таблицы. И есть реальная опасность – номерная емкость пула транзакций имеет свой предел (~ 2 в степени 32 или 4 млрд номеров). Что произойдет, если произойдет переполнение счетчика транзакций? Получится, что записи, сделанные старыми транзакциями, окажутся в будущем и не будут видны новым транзакциям. Этого допустить нельзя. Надо как-то выходить из ситуации. Разработчики PostgreSQL вышли из ситуации таким образом. Были введены специальные номера транзакций, которые всегда находятся в прошлом, относительно номеров обычных транзакций. Была создана процедура, которая системные поля записей xmin заменяет на специальный номер транзакции (FrozenXID), таким образом старые записи всегда будут находится в прошлом для всех новых транзакций.

9. Зачем нужны хранимые процедуры и какие особенности их использования?

**Храни́мая процеду́ра** — объект базы данных, представляющий собой набор SQL-инструкций, который компилируется один раз и хранится на сервере. Хранимые процедуры очень похожи на обыкновенные процедуры языков высокого уровня, у них могут быть входные и выходные параметры и локальные переменные, в них могут производиться числовые вычисления и операции над символьными данными, результаты которых могут присваиваться переменным и параметрам. В хранимых процедурах могут выполняться стандартные операции с базами данных (как DDL, так и DML). Кроме того, в хранимых процедурах возможны циклы и ветвления, то есть в них могут использоваться инструкции управления процессом исполнения.

Хранимые процедуры похожи на определяемые пользователем функции (UDF). Основное различие заключается в том, что пользовательские функции можно использовать как и любое другое выражение в SQL запросе, в то время как хранимые процедуры должны быть вызваны с помощью функции CALL:

```
CALL процедура(…)
```

или

```
EXECUTE процедура(…)
```

Хранимые процедуры могут возвращать множества результатов, то есть результаты запроса SELECT. Такие множества результатов могут обрабатываться, используя курсоры, другими сохранёнными процедурами, возвращая указатель результирующего множества, либо же приложениями. Хранимые процедуры могут также содержать объявленные переменные для обработки данных и курсоров, которые позволяют организовать цикл по нескольким строкам в таблице. Стандарт SQL предоставляет для работы выражения IF, LOOP, REPEAT, CASE и многие другие. Хранимые процедуры могут принимать переменные, возвращать результаты или изменять переменные и возвращать их, в зависимости от того, где переменная объявлена.

Хранимые процедуры позволяют повысить производительность, расширяют возможности программирования и поддерживают функции безопасности данных.

Вместо хранения часто используемого запроса, клиенты могут ссылаться на соответствующую хранимую процедуру. При вызове хранимой процедуры её содержимое сразу же обрабатывается сервером.

В большинстве СУБД при первом запуске хранимой процедуры она компилируется (выполняется синтаксический анализ и генерируется план доступа к данным). В дальнейшем её обработка осуществляется быстрее. В СУБД Oracle выполняется интерпретация хранимого процедурного кода, сохраняемого в словаре данных.

10. Зачем нужны триггера  и какие особенности их использования?

**Три́ггер** (англ. *trigger*) — хранимая процедура особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено действием по модификации данных: добавлением `INSERT`, удалением `DELETE` строки в заданной таблице, или изменением `UPDATE` данных в определённом столбце заданной таблицы реляционной базы данных. Триггеры применяются для обеспечения целостности данных и реализации сложной бизнес-логики. Триггер запускается сервером автоматически при попытке изменения данных в таблице, с которой он связан. Все производимые им модификации данных рассматриваются как выполняемые в транзакции, в которой выполнено действие, вызвавшее срабатывание триггера. Соответственно, в случае обнаружения ошибки или нарушения целостности данных может произойти откат этой транзакции.

Момент запуска триггера определяется с помощью ключевых слов `BEFORE` (триггер запускается до выполнения связанного с ним события; например, до добавления записи) или `AFTER` (после события). В случае, если триггер вызывается до события, он может внести изменения в модифицируемую событием запись (конечно, при условии, что событие — не удаление записи). Некоторые СУБД накладывают ограничения на операторы, которые могут быть использованы в триггере (например, может быть запрещено вносить изменения в таблицу, на которой «висит» триггер, и т. п.).

Кроме того, триггеры могут быть привязаны не к таблице, а к представлению (VIEW). В этом случае с их помощью реализуется механизм «обновляемого представления». В этом случае ключевые слова `BEFORE` и `AFTER` влияют лишь на последовательность вызова триггеров, так как собственно событие (удаление, вставка или обновление) не происходит.

В некоторых серверах триггеры могут вызываться не для каждой модифицируемой записи, а один раз на изменение таблицы. Такие триггеры называются табличными.

Пример (Oracle Database):

```
/* Триггер на уровне таблицы */
CREATE OR REPLACE TRIGGER DistrictUpdatedTrigger
AFTER UPDATE ON district
BEGIN
 insert into info values ('table "district" has changed');
```

```
END;
```

В этом случае для отличия табличных триггеров от строчных вводится дополнительные ключевые слова при описании строчных триггеров. В Oracle это словосочетание FOR EACH ROW.

Пример:

```
/* Триггер на уровне строки */
CREATE OR REPLACE TRIGGER DistrictUpdatedTrigger
AFTER UPDATE ON district FOR EACH ROW
BEGIN
 insert into info values ('one string in table "district" has changed');
END;
```

11. Что такое журнал транзакций? Для чего он нужен и как работает?

**Журнализация изменений** — функция СУБД, которая сохраняет информацию, необходимую для восстановления базы данных в предыдущее согласованное состояние в случае логических или физических отказов.

В простейшем случае журнализация изменений заключается в последовательной записи во внешнюю память всех изменений, выполняемых в базе данных. Записывается следующая информация:

- порядковый номер, тип и время изменения;
- идентификатор транзакции;
- объект, подвергшийся изменению (номер хранимого файла и номер блока данных в нём, номер строки внутри блока);
- предыдущее состояние объекта и новое состояние объекта.

Формируемая таким образом информация называется **журнал изменений** базы данных. Журнал содержит отметки начала и завершения транзакции, и отметки принятия контрольной точки (см. ниже).

В СУБД с отложенной записью блоки данных внешней памяти снабжаются отметкой порядкового номера последнего изменения, которое было выполнено над этим блоком данных. В случае сбоя системы эта отметка позволяет узнать какая версия блока данных успела достичь внешней памяти.

СУБД с отложенной записью периодически выполняет контрольные точки. Во время выполнения этого процесса все незаписанные данные переносятся на внешнюю память, а в журнал пишется отметка принятия контрольной точки. После этого содержимое журнала, записанное до контрольной точки может быть удалено.

Журнал изменений может не записываться непосредственно во внешнюю память, а аккумулироваться в оперативной. В случае подтверждения транзакции СУБД дожидается записи оставшейся части журнала на внешнюю память. Таким образом гарантируется, что все данные, внесённые после сигнала подтверждения, будут перенесены во внешнюю память, не дожидаясь переписи всех измененных блоков из дискового кэша. СУБД дожидается записи оставшейся части журнала так же при выполнении контрольной точки.

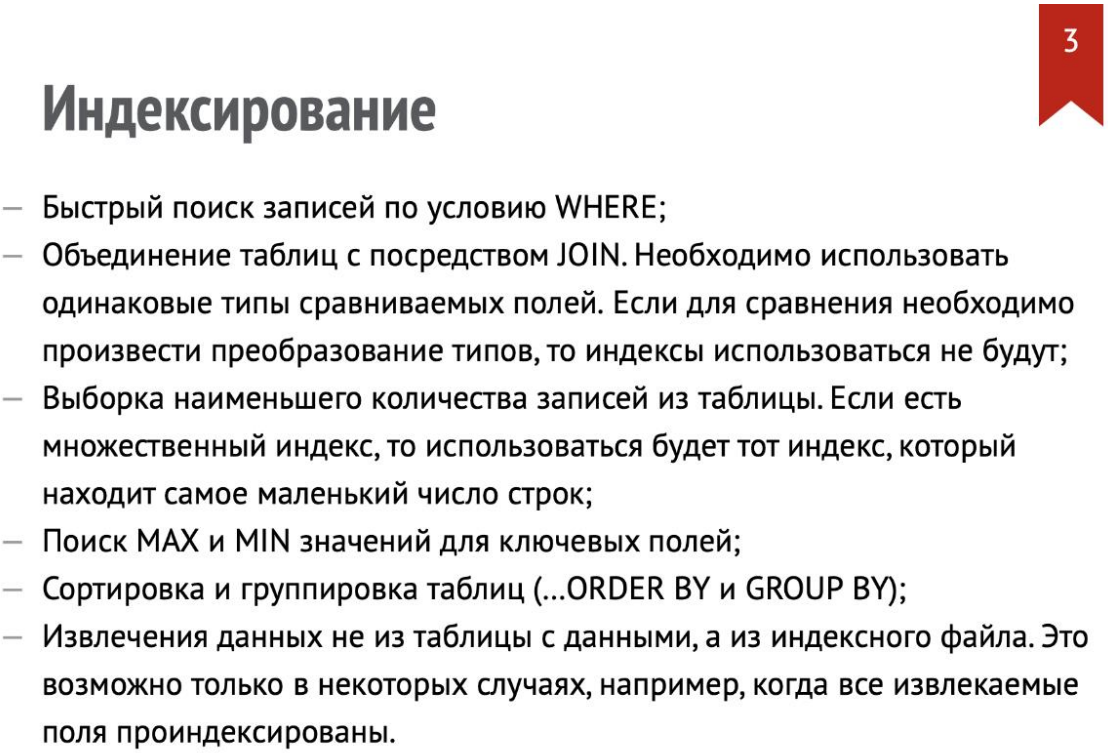
**В случае логического отказа** или сигнала отката одной транзакции журнал сканируется в обратном направлении, и все записи отменяемой транзакции извлекаются из журнала вплоть до отметки начала транзакции. Согласно извлеченной информации выполняются действия, отменяющие действия транзакции, а в журнал записываются **компенсирующие записи**. Этот процесс называется **откат** (rollback).

**В случае физического отказа**, если ни журнал, ни сама база данных не повреждена, то выполняется процесс **прогонки** (rollforward). Журнал сканируется в прямом направлении, начиная от предыдущей контрольной точки. Все записи извлекаются из журнала вплоть до конца журнала. Извлеченная из журнала информация вносится в блоки данных внешней памяти, у которых отметка номера изменений меньше, чем записанная в журнале. Если в процессе прогонки снова возникает

сбой, то сканирование журнала вновь начнется сначала, но фактически восстановление продолжится с той точки, откуда оно прервалось.

12. Зачем нужны индексы?

**Индекс** (англ. *index*) — объект базы данных, создаваемый с целью повышения производительности поиска данных. Таблицы в базе данных могут иметь большое количество строк, которые хранятся в произвольном порядке, и их поиск по заданному критерию путём последовательного просмотра таблицы строка за строкой может занимать много времени. Индекс формируется из значений одного или нескольких столбцов таблицы и указателей на соответствующие строки таблицы и, таким образом, позволяет искать строки, удовлетворяющие критерию поиска. Ускорение работы с использованием индексов достигается в первую очередь за счёт того, что индекс имеет структуру, оптимизированную под поиск — например, сбалансированного дерева.




13. Что такое частичные индексы?

*Частичный индекс* — это индекс, который строится по подмножеству строк таблицы, определяемому условным выражением (оно называется *предикатом* частичного индекса). Такой индекс содержит записи только для строк, удовлетворяющих предикату. Частичные индексы довольно специфичны, но в ряде ситуаций они могут быть очень полезны.

Частичные индексы могут быть полезны, во-первых, тем, что позволяют избежать индексирования распространённых значений. Так как при поиске распространённого значения (такого, которое содержится в значительном проценте всех строк) индекс всё равно не будет использоваться, хранить эти строки в индексе нет смысла. Исключив их из индекса, можно уменьшить его размер, а значит и ускорить запросы, использующие этот индекс.

Во-вторых, частичные индексы могут быть полезны тем, что позволяют исключить из индекса значения, которые обычно не представляют интереса;

Третье возможное применение частичных индексов вообще не связано с использованием индекса в запросах. Идея заключается в том, чтобы создать уникальный индекс по подмножеству строк таблицы.

Примеры:

https://postgrespro.ru/docs/postgrespro/9.5/indexes-partial

14. Что такое кластерные индексы?

Кластерный индекс — это древовидная структура данных, при которой значения индекса хранятся вместе с данными, им соответствующими. И индексы, и данные при такой организации упорядочены. При добавлении новой строки в таблицу, она дописывается не в конец файла*, не в конец плоского списка, а в нужную ветку древовидной структуры, соответствующую ей по сортировке.

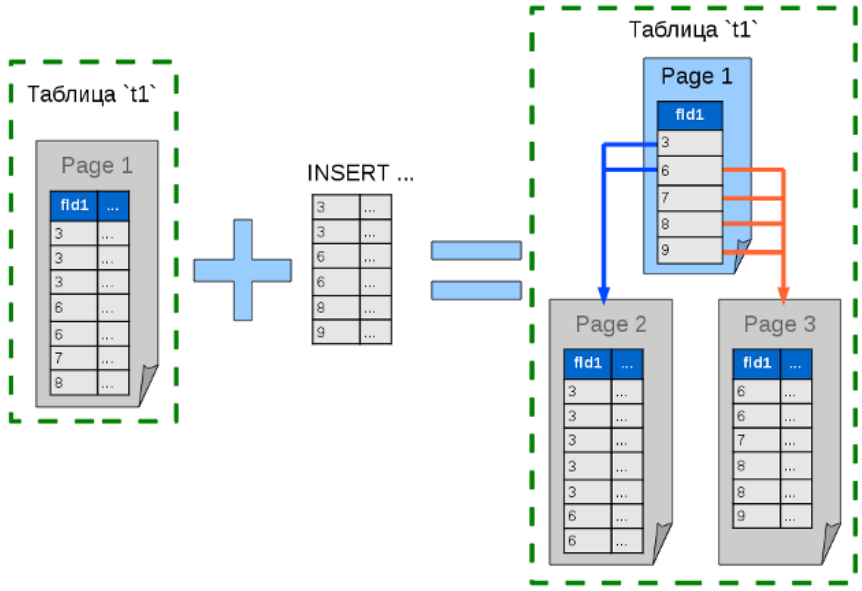
## Кластерный индекс

```
CREATE TABLE movies (
    id SERIAL PRIMARY KEY,
    title TEXT NOT NULL
);

CLUSTER movies USING movies_pkey;
CLUSTER movies;
CLUSTER;
```

Кластерный индекс (или кластерный ключ) сохраняет не только значения колонки в отсортированном виде, а и данные всей строки.

Это позволяет минимизировать количество операций чтения с диска при работе с таким индексом. В таблице может быть только один кластерный индекс.




По мере добавления всё новых и новых данных, дерево будет усложняться и углубляться. И чем больше оно будет и ветвистее, тем больший выйгрышь даст такая схема хранения данны.

15. Что такое покрывающие индексы?

Покрывающий (covering) индекс - это индекс, которого достаточно для ответа на запрос вовсе без обращения к самой таблице. В самом индексе хранится достаточно данных для ответа на запрос и, хоть и возможно по индексу достать всю строку данных - это просто не нужно. За счёт того, что не нужно дёргать непосредственно таблицу, а ответить можно используя только индекс - покрывающие индексы несколько быстрее (насколько - зависит от дисков, кэша и объёма горячей части базы). Но при этом, разумеется, сам индекс становится больше и злоупотреблять этим не нужно.

17

## Покрывающий индекс

```
CREATE TABLE movies (
    id SERIAL PRIMARY KEY,
    title TEXT NOT NULL
);

CREATE INDEX idx_movies_title ON movies (title);

SELECT title FROM movies
WHERE title = 'Alice in Wonderland';
```

Покрывающий индекс содержит все данные, необходимые для выполнения запроса.

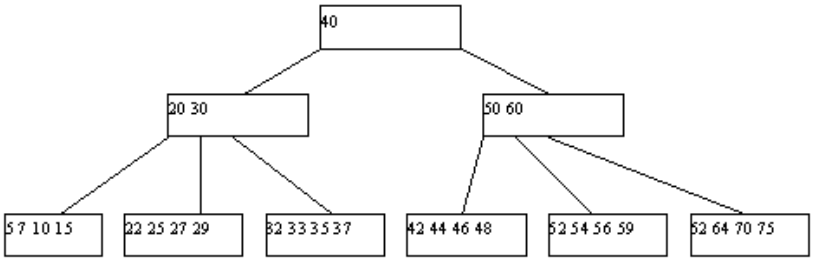
В PostgreSQL индексы не хранят информацию о видимости записи для MVCC. Из-за этого они могут быть покрывающими только если в таблице нет мертвых кортежей.

16. В чем разница между B-Tree/Hash/Bitmap индексами?

https://bozaro.github.io/tech-db-lectures/05/#4

### *B-Tree*

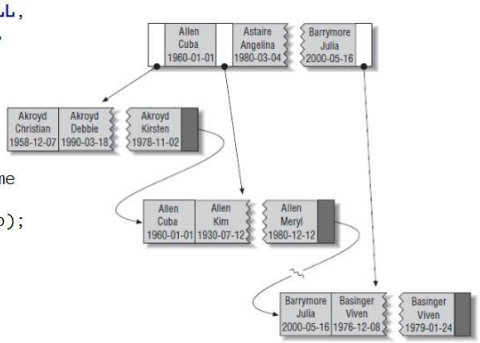
Семейство B-Tree индексов — это наиболее часто используемый тип индексов, организованных как сбалансированное дерево, упорядоченных ключей. Они поддерживаются практически всеми СУБД как реляционными, так нереляционными, и практически для всех типов данных.




## Индексирование btree (пример)

```
CREATE TABLE people (
    last_name TEXT NOT NULL,
    first_name TEXT NOT NULL,
    dob TIMESTAMP NOT NULL,
    gender INT NOT NULL
);

CREATE INDEX idx_people_name
ON people USING btree
(last_name, first_name, dob);
```




## Индексирование btree (особенности)

**Можно:**

- Поиск по полному значению;
- Поиск по самому левому префиксу;
- Поиск по префиксу столбца;
- Поиск по диапазону значений;
- Поиск по полному совпадению одной части и диапазону в другой части;
- Запросы только по индексу.

**Нельзя:**

- Поиск без использования левой части ключа;
- Нельзя пропускать столбцы;
- Оптимизация после поиска в диапазоне.

### *HASH*

Hash-индексы были предложены Артуром Фуллером, и предполагают хранение не самих значений, а их хэшей, благодаря чему уменьшается размер(а, соответственно, и увеличивается скорость их обработки) индексов из больших полей. Таким образом, при запросах с использованием HASH-индексов, сравниваться будут не искомое со значения поля, а хэш от искомого значения с хэшами полей.

Из-за нелинейнойсти хэш-функций данный индекс нельзя сортировать по значению, что приводит к невозможности использования в сравнениях больше/меньше и «is null». Кроме того, так как хэши не уникальны, то для совпадающих хэшей применяются методы разрешения коллизий.

```
CREATE TEMPORARY TABLE testhash (
    fname TEXT NOT NULL,
    lname TEXT NOT NULL
);
CREATE INDEX idx_testhash_fname
ON testhash USING hash (fname);
```

| fname | lname |
|---|---|
| Arjen | Lentz |
| Baron | Schwartz |
| Peter | Zaitsev |
| Vadim | Tkachenko |

```
f('Arjen') = 2323
f('Baron') = 7437
f('Peter') = 8784
f('Vadim') = 2458
```

| Ячейка | Значение |
|---|---|
| 2323 | Указатель на строку 1 |
| 2458 | Указатель на строку 4 |
| 7437 | Указатель на строку 2 |
| 8784 | Указатель на строку 3 |

```
SELECT lname FROM testhash WHERE fname = 'Peter';
```

## Индексирование hash (особенности)

- Нельзя использовать данные в индексе, чтобы избежать чтения строк.
- Нельзя использовать для сортировки, поскольку строки в нем не хранятся в отсортированном порядке.
- Хеш-индексы не поддерживают поиск по частичному ключу, так как хеш-коды вычисляются для всего индексируемого значения.
- Хеш-индексы поддерживают только сравнения на равенство, использующие операторы =, IN() и <=>.
- Доступ к данным в хеш-индексе очень быстр, если нет большого количества коллизий.
- Некоторые операции обслуживания индекса могут оказаться медленными, если количество коллизий велико.

**Важно**
В PostgreSQL до 10 версии hash-индекс не записывается в WAL-лог, т. е. он не транзакционен.

### *Bitmap*

Bitmap index – метод битовых индексов заключается в создании отдельных битовых карт (последовательность 0 и 1) для каждого возможного значения столбца, где каждому биту соответствует строка с индексируемым значением, а его значение равное 1 означает, что запись, соответствующая позиции бита содержит индексируемое значение для данного столбца или свойства.

**Данные**

| id | name | gender |
|---|---|---|
| 1 | Иван | Мужской |
| 2 | Евгений | Мужской |
| 3 | Александра | Женский |
| 4 | Петр | Мужской |
| 5 | Мария | Женский |

**Битовые маски**

| value | first-id | bitmask |
|---|---|---|
| Женский | 1 | 00101 |
| Мужской | 1 | 11010 |

17. В чем разница между Merge join/Nested loop join/Hash join?

**MERGE JOIN**

Соединение двух отсортированных последовательностей.

Работает быстро и за один проход обоих списков.

**HASH JOIN**

Меньшее отношение помещается в хэш-таблицу. Затем для каждой строки из большей таблицы выполняется поиск значений, соответствующих условию соединения.

Соединение только по условию эквивалентности.

**NESTED LOOP**

Соединение вложенными циклами.

18. План выполнения запросов. EXPLAIN.

Когда вы выполняете какой-нибудь запрос, оптимизатор запросов MySQL пытается придумать оптимальный план выполнения этого запроса. Вы можете посмотреть этот самый план используя запрос с ключевым словом EXPLAIN.

Использовать оператор EXPLAIN просто. Его необходимо добавлять в запросы перед оператором SELECT. Давайте проанализируем вывод, чтобы познакомиться с информацией, возвращаемой командой.

```
EXPLAIN SELECT * FROM categories
```

https://habr.com/ru/post/211022/

- EXPLAIN ничего не говорит о том, как влияют на запрос триггеры и пользовательские функции.
- Не работает с хранимыми процедурами, хотя можно разложить процедуру на отдельные запросы и вызвать EXPLAIN для каждого из них.
- Часть отображаемой статистической информации – всего лишь оценка, иногда очень неточная.
- Для получения фактически затраченного времени, можно выполнить EXPLAIN ANALYZE.

19. Нормализация и денормализация данных.

**https://itif.ru/osnovy-relyacionnyx-bd/**

13

## Нормализация

- Нормализованные таблицы обычно обновляются быстрее, чем ненормализованные.
- Когда данные хорошо нормализованы, они либо редко дублируются, либо не дублируются совсем. Так что изменять приходится меньше данных.
- Нормализованные таблицы обычно меньше по размеру, поэтому лучше помещаются в памяти и их производительность выше.
- Из-за отсутствия избыточных данных реже возникает необходимость в запросах с фразами DISTINCT или GROUP BY для извлечения списков значений.

## Денормализация

**Денормализация**

Намеренное приведение структуры базы данных в состояние, не соответствующее критериям нормализации, обычно проводимое с целью ускорения операций чтения из базы за счет добавления избыточных данных.

- Обновление данных триггерах.
- Обновление данных по расписанию.
- Инкрементальное обновление данных.

15

## Нормализация/денормализация

```
alter table movies
  add column rating_sum float default 0 not null,
  add column rating_cnt int default 0 not null;

update movies m
set
  rating_cnt = r.rating_cnt,
  rating_sum = r.rating_sum
from (
  select movie_id, count(*) as rating_cnt, sum(rating) as rating_sum
  from ratings
  group by movie_id
) r where (m.id = r.movie_id);

create trigger ...;
```

## Нормализация/денормализация

**После денормализации (12 msec):**

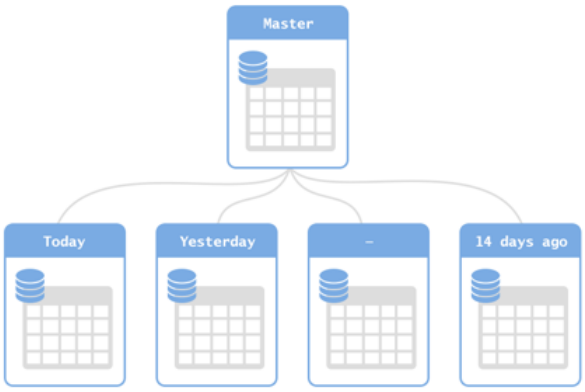
```
select m.id, m.title, m.rating_sum / m.rating_cnt
from movies       m
join movie_genres gm on (gm.movie_id = m.id)
join genres       g  on (g.id = gm.genre_id)
join movie_tags   tm on (tm.movie_id = m.id)
join tags         t  on (t.id = tm.tag_id)
where lower(g.name) = lower('Comedy')
and lower(t.name) like lower('Zombie%')
and m.rating_cnt > 0
group by m.id
order by 3 desc
```

20. Что такое секционирование и для чего оно нужно?

Пример: https://habr.com/ru/post/74984/

### Секционирование (англ. partitioning)

Разделение хранимых объектов баз данных (таких как таблиц, индексов, материализованных представлений) на отдельные части с раздельными параметрами физического хранения.



21. Что такое COLLATION?

Определяет параметры сортировки базы данных или столбца таблицы либо операцию приведения параметров сортировки при использовании с выражением строки символов. Именем параметров сортировки может быть либо имя параметров сортировки Windows, либо имя параметров сортировки SQL. Если не указывать этот параметр при создании базы данных, ей будут назначены параметры сортировки по умолчанию для экземпляра SQL Server. Если не указывать его при создании столбца таблицы, столбцу назначаются параметры сортировки по умолчанию для базы данных.

Синтаксис

```
COLLATE { <collation_name> | database_default }
<collation_name> :: =
    { Windows_collation_name } | { SQL_collation_name }
```

Аргументы

*collation_name* — имена параметров сортировки, применяемых к выражению, определению столбца или определению базы данных. Аргументом *collation_name* может быть только заданное имя *Windows_collation_name* или *SQL_collation_name*. Аргумент *collation_name* должен быть литеральным значением. Имя *collation_name* не может быть представлено переменной или выражением.

Аргумент *Windows_collation_name* является именем параметров сортировки для Windows Collation Name.

Аргумент *SQL_collation_name* является именем параметров сортировки для SQL Server Collation Name.

**database_default** — заставляет предложение COLLATE наследовать параметры сортировки текущей базы данных.

https://docs.microsoft.com/ru-ru/sql/t-sql/statements/collations?view=sql-server-2017

22. Чем отличается коррелирующие и не коррелирующие подзапросы?

Коррелирующие подзапросы позволяют иногда очень кратко написать запросы, которые могут выглядеть весьма громоздко при использовании других языковых средств.

коррелирующий подзапрос — это подзапрос, который содержит ссылку на столбцы из включающего его запроса (назовем его основным). Таким образом, коррелирующий подзапрос будет выполняться для каждой строки основного запроса, так как значения столбцов основного запроса будут меняться.

Некоррелирующий подзапрос: данные, выбираемые некоррелированным подзапросом, никак не зависят от внешнего запроса.

# SUBQUERIES vs JOIN

### Коррелирующий подзапрос

```
SELECT E.*
FROM Employee E WHERE EXISTS (
  SELECT *
  FROM Department D WHERE D.DepartmentID = E.DepartmentID
);
```

### Не коррелирующий подзапрос

```
SELECT E.*
FROM Employee E WHERE E.DepartmentID IN (
  SELECT DepartmentID
  FROM Department D
);
```

### JOIN

```
SELECT E.*
FROM Employee E
JOIN Department D ON (E.DepartmentID = D.DepartmentID);
```

23. Что лучше, JSON или подзапрос? Почему?

Часто требуется выбрать дополнительные параметры плюсом к основной выборке. Это можно сделать двумя способами:

- LEFT JOIN к основному запросу
- Подзапрос в секции SELECT

Пример запроса, выбирающего номенклатуру и товарную группу номенклатуры:

Подзапрос в секции SELECT:

```
SELECT nom.Наименование,
(SELECT tg.Наименование FROM Товарная группа tg WHERE tg.ID = nom.Товарная группа)
FROM Номенклатура nom
```

Вариант с LEFT JOIN:

```
SELECT nom.Наименование, tg.Наименование
FROM Номенклатура nom
LEFT JOIN Товарная группа tg
 ON(tg.ID = nom.Товарная группа)
```

После разбора плана выполнения запроса в MS SQL Server, было выявлено:

- При выборке в секции SELECT требуется дополнительное время на слияние основной выборки и подзапроса. Вышло 1% от общего времени.(+ LEFT JOIN / - SELECT)
- Оказалось, что каждый LEFT JOIN выполняется отдельным потоком! В то время как подзапросы выполняются последовательно после основной выборки. (+ LEFT JOIN / - SELECT)
- Каждый LEFT JOIN объединяется в результирующую выборку, что требует дополнительной памяти. В то время как подзапрос вернет нам одно значение на

каждую строку, т.е. для получения результата нам нужно меньше памяти. (- LEFT JOIN / + SELECT)

Вывод: **В условиях многоядерных серверов *LEFT JOIN имеет неоспоримые преимущества***

Ограничения в подзапросах:
При выборке дополнительных параметров в секции SELECT мы можем столкнуться с ситуацией, когда подзапрос вернул нам более одной записи. Отсечь это можно, указав принудительно число выбираемых записей:

```
SELECT nom.Наименование,
(SELECT TOP 1 tg.Наименование FROM Товарная группа tg WHERE tg.ID = nom.Товарная группа)
FROM Номенклатура nom
```

Такой запрос выполнялся дольше на 96% по сравнению с аналогичным без TOP 1

Разбор плана показал, что 1% дополнительного времени тратится вложенный цикл, а 95% на просмотр определенных строк не кластеризованного индекса!
Из этого можно сделать один вывод: никогда не используйте без надобности ограничения в подзапросах.

*В заключении* хотел бы сказать, что бывают ситуации когда без подзапросов не обойтись (группировки, сортировки и т.д.), но использовать их нужно обосновано, если есть возможность, **то лучше пользоваться LEFT JOIN.**

**Если кратко своими словами: подзапросы вычисляются для каждой строки основного запроса (от внутреннего к внешнему). Join же просто соединяет результаты подзапросов, не делая вычисления для каждой строки.**

24. Почему использование индекса может замедлить выполнения запроса?

- Увеличение числа индексов замедляет операции добавления, обновления, удаления строк таблицы, поскольку при этом приходится обновлять сами индексы.
- Индексы занимают дополнительный объем памяти, поэтому перед созданием индекса следует убедиться, что планируемый выигрыш в производительности запросов превысит дополнительную затрату ресурсов компьютера на сопровождение индекса.

25. Что такое селективность индекса?

**Селективность индекса** – это показатель того, сколько строк от общего числа приходится на одно ключевое значение индекса. Построим формулу.

```
Селективность индекса = ( NUM_ROWS/DISTINCT_KEY) / NUM_ROWS = 1/DISTINCT_KEY
```

Таким образом, чтобы рассчитать селективность индекса довольно посмотреть значение DISTINCT_KEYS в динамическом представлении ALL_INDEXES. Селективность чаще вычисляют в процентах. Чем больше этот процент, тем меньше (хуже) селективность. Селективность хороша, если мало строк имеют одинаковые ключевые значения.

26. Какие есть варианты протоколирования запросов?

**Конфигурация:**

```
log_duration = on
log_min_duration_statement = 50
```

**Запрос:**

```
set log_min_duration_statement = 50;
select * from movies where title = 'Alice in Wonderland';
```

**Пример:**

```
2017-03-12 22:34:32 MSK [8960-5] postgres@movielens LOG:
duration: 50.157 ms  statement: select * from movies where
title = 'Alice in Wonderland'
2017-03-12 22:35:42 MSK [8960-6] postgres@movielens LOG:
duration: 54.305 ms  statement: select * from movies where
title = 'Alice in Wonderland'
```

**Конфигурация:**

```
log_duration = on
log_lock_waits = on
log_min_duration_statement = 50
log_filename = 'postgresql-%Y-%m-%d_%H%M%S'
log_directory = '/var/log/postgresql'
log_destination = 'csvlog'
logging_collector = on
```

Логирование в CSV создаёт файлы в формате, пригодном для анализа утилитами вида pgbadger:

```
sudo apt instal libtext-csv-xs-perl pgbadger
pgbadger /var/log/*.csv
```

# Протоколирование запросов

- Данные или индексы могли к тому моменту еще отсутствовать в кэше. Это обычный случай, когда сервер СУБД только запущен или не настроен должным образом.
- Мог идти ночной процесс резервного копирования, из-за чего все операции дискового ввода/вывода замедлялись.
- Сервер мог обрабатывать в тот момент другие запросы, поэтому данный выполнялся медленнее.

**Долго выполняющиеся запросы**

Периодические выполняемые пакетные задания действительно могут запускать долго выполняющиеся запросы, но обычные запросы не должны занимать много времени.

**Запросы, больше всего нагружающие сервер**

Ищите запросы, которые потребляют большую часть времени сервера. Напомним, что короткие запросы, выполняемые очень часто, тоже могут занимать много времени.

**Новые запросы**

Ищите запросы, которых вчера не было в первой сотне, а сегодня они появились. Это могут быть новые запросы или запросы, которые обычно выполнялись быстро, а теперь замедлились из-за изменившейся схемы индексации. Либо произошли еще какие-то изменения.

27. Что такое VACUUM?

VACUUM — провести сборку мусора и, возможно, проанализировать базу данных

## Синтаксис

```
VACUUM [ ( { FULL | FREEZE | VERBOSE | ANALYZE } [, ...] ) ] [ имя_таблицы [ (имя_столбца [, ...] ) ] ]

VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ имя_таблицы ]

VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ имя_таблицы [ (имя_столбца [, ...] ) ] ]
```

## Описание

VACUUM высвобождает пространство, занимаемое «мёртвыми» кортежами. При обычных операциях Postgres Pro кортежи, удалённые или устаревшие в результате обновления, физически не удаляются из таблицы; они сохраняются в ней, пока не будет выполнена команда VACUUM. Таким образом, периодически необходимо выполнять VACUUM, особенно для часто изменяемых таблиц.

*Параметр INCLUDE* позволяет указать неключевые столбцы, которые добавляются к страницам узлов некластеризованного индекса. Имена столбцов в списке INCLUDE не должны повторяться, и столбец нельзя использовать одновременно как ключевой и неключевой.

Чтобы по-настоящему понять полезность параметра INCLUDE, нужно понимать, что собой представляет *покрывающий индекс (covering index)*. Если все столбцы запроса включены в индекс, то можно получить значительное повышение производительности, т.к. оптимизатор запросов может определить местонахождение всех значений столбцов по страницам индекса, не обращаясь к данным в таблице. Такая возможность называется покрывающим индексом или покрывающим запросом. Поэтому включение в страницы узлов некластеризованного индекса дополнительных неключевых столбцов позволит получить больше покрывающих запросов, при этом их производительность будет значительно повышена.

Оптимально ли составлен запрос? Какие индексы нужно построить для эффективного выполнения запроса и почему?

https://gitlab.com/snippets/1846999

## 1

```sql
SELECT rental_id
FROM rental
WHERE rental_date BETWEEN '2005-07-01' AND '2005-08-01'
  AND customer_id = 2
ORDER BY rental_date, inventory_id;
```

```sql
CREATE INDEX idx ON rental (customer_id, rental_date, inventory_id) INCLUDE (rental_id);
```

## 2

```sql
SELECT rental_id
FROM rental
WHERE EXTRACT(EPOCH FROM '2005-07-01'::timestamp - rental_date) > 0
  AND customer_id = 2
ORDER BY rental_date;
```

```sql
SELECT rental_id
FROM rental
WHERE rental_date < '2005-07-01'
  AND customer_id = 2
ORDER BY rental_date;

CREATE INDEX idx ON rental (customer_id, rental_date) INCLUDE (rental_id);
```

## 3

```
SELECT rental_id
FROM rental
WHERE rental_date BETWEEN '2005-07-01' AND '2005-08-01'
  AND customer_id IN (2, 42, 73);
```

```
CREATE INDEX idx ON rental (customer_id, rental_date) INCLUDE (rental_id);
```

## 4

```
SELECT city.city_id, city.city, country.country
FROM city
JOIN country ON city.country_id = country.country_id
WHERE LOWER(LEFT(city.city, 2)) = 'mo';
```

```
CREATE INDEX idx ON country(country_id);
CREATE INDEX idx2 ON city ((LOWER(LEFT(city, 2))));
```

## 5

### До

```
SELECT COUNT(*), country.country
FROM city
JOIN country ON city.country_id = country.country_id
GROUP BY country.country;
```

### После

```
SELECT COUNT(*), country.country
FROM country
JOIN city ON city.country_id = country.country_id
GROUP BY country.country_id;

CREATE INDEX idx ON country(country_id);
```

# 6

## До

```
SELECT film.title, SUM(payment.amount)
FROM customer
JOIN rental ON (rental.customer_id = customer.customer_id)
JOIN inventory ON (rental.inventory_id = inventory.inventory_id)
JOIN film ON (inventory.film_id = film.film_id)
JOIN payment ON (payment.rental_id = rental.rental_id)
WHERE EXTRACT(YEAR FROM rental.rental_date) = 2005
  AND customer.store_id = 2
GROUP BY film.title;
```

## После

```
SELECT film.title, SUM(payment.amount)
FROM customer
JOIN rental ON (rental.customer_id = customer.customer_id)
JOIN inventory ON (inventory.inventory_id = rental.inventory_id)
JOIN film ON (film.film_id = inventory.film_id)
JOIN payment ON (payment.rental_id = rental.rental_id)
WHERE EXTRACT(YEAR FROM rental.rental_date) = 2005
  AND customer.store_id = 2
GROUP BY film.title;
```

```
-- CREATE INDEX customer_idx_id (store_id) ON customer;
-- CREATE INDEX film_idx_id_title (film_id,title) ON film;
-- CREATE INDEX inventory_idx_id (inventory_id) ON inventory;
-- CREATE INDEX payment_idx_id (rental_id) ON payment;
-- CREATE INDEX rental_idx_id (customer_id) ON rental;
CREATE INDEX idx ON rental(rental_date);
CREATE INDEX idx2 ON customer(store_id);
```

# 7

```
SELECT country.country, COUNT(*)
FROM store
JOIN address ON (store.address_id = address.address_id)
JOIN city ON (city.city_id = address.city_id)
JOIN country ON (country.country_id = city.country_id)
WHERE LOWER(LEFT(country.country, 1)) = 'a'
GROUP BY country.country;
```

```
SELECT country.country, COUNT(*)
FROM store
JOIN address ON (store.address_id = address.address_id)
JOIN city ON (city.city_id = address.city_id)
JOIN country ON (country.country_id = city.country_id)
WHERE LOWER(LEFT(country.country, 1)) = 'a'
GROUP BY country.country_id; --zpynn no id

CREATE INDEX idx ON country(LOWER(LEFT(country, 1)), country_id);
-- CREATE INDEX address_idx_id (city_id) ON address;
-- CREATE INDEX city_idx_id (country_id) ON city;
-- CREATE INDEX country_idx_country (country) ON country;
-- CREATE INDEX store_idx_id (address_id) ON store;
```

# 8

```
SELECT film_id, title, description
FROM film
WHERE (
    SELECT COUNT(*)
    FROM actor
    JOIN film_actor ON (film_actor.actor_id = actor.actor_id)
    WHERE film_actor.film_id = film.film_id
      AND LOWER(actor.last_name) = 'monroe'
) > 0
  AND film.rating = 'G'
  AND film.release_year BETWEEN 2005 AND 2008;
```

Переписать через JOIN

```
SELECT film.film_id, film.title, film.description
FROM film
JOIN film_actor ON film_actor.film_id = film.film_id
JOIN actor ON actor.actor_id = film_actor.actor_id
WHERE LOWER(actor.last_name) = 'monroe'
  AND film.rating = 'G'
  AND film.release_year BETWEEN 2004 AND 2008;

CREATE INDEX idx ON actor(LOWER(actor.last_name));
CREATE INDEX idx2 ON film(rating, release_year);
```