1) *mysql -h localhost -u root -p pass* - вход
2) *CREATE DATABASE vova;*
3) *CREATE [TEMPORARY - временная(только в сессию)] TABLE <name_table> [create_definition-всякие поля и свойства(типо идексы и ключи)]*

**[PRIMARY KEY]** - Задает первичный ключ таблицы.
В таблице может быть задан только одни первичный ключ. Все значения столбца, помеченного как перичный ключ, не должны содержать значения NULL. Если при создании таблицы первичный ключ явно указан не был, а приложение его запрашивает, то БД MySQL автоматически устанавливает первый столбец с параметром *UNIQUE*, если во всех значениях этого столбца нигде не встречается значение NULL. Может состоять из нескольких полей

Поля *PRIMARY KEY* являются проиндексированными полями

**[KEY]**
- Является синонимом к INDEX

**[INDEX]**
- Задает поля, которые будут проиндексированы.
Индексация полей полезна для ускорения работы команды **SELECT** (причем ускорение порой бывает очень даже хорошее).
Если задать поля, которые должны быть проиндексированы, то MySQL создает специальное хранилище, в котором содержатся все значения проиндесированных полей таблицы и точное местоуказание, где это значение находится. Т.е. поиск значения происходит практически мгновенно, что несомненно сказывается на скорости выполнения скрипта.
В **MySQL** проиндексированы могут быть поля любого типа. Для ускорения работы в полях типа *CHAR* и *VARCHAR* можно индексировать только несколько первых символов.

***ПРИМЕР:***

```
CREATE TABLE
        users (
        name CHAR(200) NOT NULL ,
        age SMALLINT(3),
        INDEX (name(12))
        );
```

создает таблицу с индексом по 12 первым символам.
***show index from users;*** - посмотреть индексы

## *[UNIQUE]*

- Этот ключ указывает на то, что данный столбец может иметь только уникальные значения. При попытке добавления повторяющегося значения в таблицу в поле с ключом *UNIQUE*, эта операция завершится ошибкой.

```
CREATE TABLE
   `users` (
      `name` VARCHAR(200) NOT NULL,
      `address` VARCHAR(255) NOT NULL,
      UNIQUE(`name`, `address`)
   )
```

## [FULLTEXT]

- Задает поля, к которым в последствии может быть применен полнотекстный поиск. Полнотекстный поиск является средством MySQL, направленным на поиск нужной информации в базе данных и выводе результатов в соответствии с релевантностью найденных строк относительно поискового запроса.
Полнотестный поиск введен в **MySQL** начиная с версии 3.23.23 для таблиц типа *MyISAM* и только для полей типа *VARCHAR* и *TEXT*.
При индексировании полей с ключом *FULLTEXT* происходит индексация всего значения, а не его части (т.е. задать для индексации первые n-символов НЕЛЬЗЯ).

## *ПРИМЕР*

```
CREATE TABLE users(
        name VARCHAR(200),
        family VARCHAR(200),
        FULLTEXT(name)
        )ENGINE='MyISAM';
```

**-** создание таблицы. MYISAM для FULLTEXT.

Вставляем записи.

```
mysql> SELECT * FROM users;
+------------------+-------------+
| name             | family      |
+------------------+-------------+
| vova             | Prihodko    |
| dasha            | ufimtseva   |
| dasha dasha dasha | ufasdimtseva |
| ura              | ivanov      |
| ura              | vladiev     |
| ura              | fam dasha   |
+------------------+-------------+
```

```
SELECT * FROM users WHERE MATCH (name) AGAINST('dasha');
+------------------+-------------+
| name             | family      |
+------------------+-------------+
| dasha            | ufimtseva   |
| dasha dasha dasha | ufasdimtseva |
+------------------+-------------+
```

сортирует по релевантности(здесь первая запись выше, так как там меньше слов в принципе)

```
mysql> SELECT MATCH(name) AGAINST ('dasha') FROM users;
+-------------------------------+
| MATCH(name) AGAINST ('dasha') |
+-------------------------------+
|                             0 |
|          0.9058732390403748 |
|          0.9058732390403748 |
|                             0 |
|                             0 |
|                             0 |
|                             0 |
+-------------------------------+
```

link: тыц

4. ***ALTER TABLE*** Оператор ALTER TABLE обеспечивает возможность

изменять структуру существующей таблицы. Например, можно добавлять или удалять столбцы, создавать или уничтожать индексы или переименовывать столбцы либо саму таблицу.
Оператор ALTER TABLE во время работы создает временную копию исходной таблицы. Требуемое изменение выполняется на копии, затем исходная таблица удаляется, а новая переименовывается. Так делается для того, чтобы в новую таблицу автоматически попадали все обновления кроме неудавшихся. Во время выполнения ALTER TABLE исходная таблица доступна для чтения другими клиентами. Операции обновления и записи в этой таблице приостанавливаются, пока не будет готова новая таблица.
Следует отметить, что при использовании любой другой опции для ALTER TABLE кроме RENAME, MySQL всегда будет создавать временную таблицу, даже если данные, строго говоря, и не нуждаются в копировании (например, при изменении имени столбца).

***ПРИМЕРЫ***

1. CREATE TABLE t1(a INT, b CHAR(10));
2. ALTER TABLE t1 RENAME t2; - меняем название таблицы
3. ALTER TABLE t2 MODIFY a TINYINT NOT NULL, CHANGE b c CHAR(20); - в таблице t2 модифицируем поле a, изменяем поле b на c со сменой типа
4. ALTER TABLE t2 ADD d REAL; - добавление поля
5. ALTER TABLE t2 ADD INDEX(d), ADD PRIMARY KEY(a);
6. ALTER TABLE t2 DROP INDEX d;
7. ALTER TABLE t2 DROP COLUMN c;

## 5. **INSERT**

- Если указывается ключевое слово LOW_PRIORITY, то выполнение данной команды INSERT будет задержано до тех пор, пока другие клиенты не завершат чтение этой таблицы. В этом случае данный клиент должен ожидать, пока данная команда вставки не будет завершена, что в случае интенсивного использования таблицы может потребовать значительного времени. В противоположность этому команда INSERT DELAYED позволяет данному клиенту продолжать

операцию сразу же. See section 6.4.4 Синтаксис оператора INSERT DELAYED. Следует отметить, что указатель LOW_PRIORITY обычно не используется с таблицами MyISAM, поскольку при его указании становятся невозможными параллельные вставки. See section 7.1 Таблицы MyISAM.

- Если в команде INSERT со строками, имеющими много значений, указывается ключевое слово IGNORE, то все строки, имеющие дублирующиеся ключи PRIMARY или UNIQUE в этой таблице, будут проигнорированы и не будут внесены. Если не указывать IGNORE, то данная операция вставки прекращается при обнаружении строки, имеющей дублирующееся значение существующего ключа. Количество строк, внесенных в данную таблицу, можно определить при помощи функции C API mysql_info().

## 6.UPDATE

**UPDATE [LOW_PRIORITY] [IGNORE] tbl_name**
**SET col_name1=expr1 [, col_name2=expr2, ...]**
**[WHERE where_definition]**
**[LIMIT #]**

## 7. SELECT

-Параметры (опции) DISTINCT, DISTINCTROW и ALL указывают, должны ли возвращаться дублирующиеся записи. По умолчанию установлен параметр (ALL), т.е. возвращаются все встречающиеся строки. DISTINCT и DISTINCTROW являются синонимами и указывают, что дублирующиеся строки в результирующем наборе данных должны быть удалены.
-При указании параметра STRAIGHT_JOIN оптимизатор будет объединять таблицы в том порядке, в котором они перечислены в выражении FROM. Применение данного параметра позволяет увеличить скорость выполнения запроса, если оптимизатор производит объединение таблиц неоптимальным образом.

-Выражение LIMIT может использоваться для ограничения количества строк, возвращенных командой SELECT. LIMIT принимает один или два числовых аргумента. Эти аргументы должны быть целочисленными константами. Если заданы два аргумента, то первый указывает на начало первой возвращаемой строки, а второй задает максимальное количество возвращаемых строк. При этом смещение начальной строки равно 0 (не 1):
mysql> SELECT * FROM table LIMIT 5,10; # возвращает строки 6-15

# ПРОЦЕДУРЫ.

Хранимая процедура - это способ инкапсуляции повторяющихся действий. В хранимых процедурах можно объявлять переменные, управлять потоками данных, а также применять другие техники программирования.
Причина их создания ясна и подтверждается частым использованием. С другой стороны, если вы поговорите с теми, кто работает с ними нерегулярно, то мнения разделятся на два совершенно противоположных фланга. Не забывайте об этом.

**За**

- Разделение логики с другими приложениями. Хранимые процедуры инкапсулируют функциональность; это обеспечивает связность доступа к данным и управления ими между различными приложениями.
- Изоляция пользователей от таблиц базы данных. Это позволяет давать доступ к хранимым процедурам, но не к самим данным таблиц.
- Обеспечивает механизм защиты. В соответствии с предыдущим пунктом, если вы можете получить доступ к данным только через хранимые процедуры, никто другой не сможет стереть ваши данные через команду SQL DELETE.
- Улучшение выполнения как следствие сокращения сетевого трафика. С помощью хранимых процедур множество запросов могут быть объединены.

**Против**

- Повышение нагрузки на сервер баз данных в связи с тем, что большая часть работы выполняется на серверной части, а меньшая - на клиентской.
- Придется много чего подучить. Вам понадобится выучить синтаксис MySQL выражений для написания своих хранимых процедур.

- Вы дублируете логику своего приложения в двух местах: серверный код и код для хранимых процедур, тем самым усложняя процесс манипулирования данными.
- Миграция с одной СУБД на другую (DB2, SQL Server и др.) может привести к проблемам.

## Пример.

mysql>DELIMITER // - *теперь // является знаком конца операции(вместо ;)*
mysql> CREATE PROCEDURE p2() BEGIN SELECT 'Hello world';END//

DROP PROCEDURE p2; - удаление процедуры

Давайте посмотрим, как можно передавать в хранимую процедуру параметры.

- CREATE PROCEDURE proc1 (): пустой список параметров
- CREATE PROCEDURE proc1 (IN varname DATA-TYPE): один входящий параметр. Слово IN необязательно, потому что параметры по умолчанию - IN (входящие).
- CREATE PROCEDURE proc1 (OUT varname DATA-TYPE): один возвращаемый параметр.
- CREATE PROCEDURE proc1 (INOUT varname DATA-TYPE): один параметр, одновременно входящий и возвращаемый.

2)Естественно, вы можете задавать несколько параметров разных типов.

```
DELIMITER //
CREATE PROCEDURE proc(IN var INT)
BEGIN
    SELECT var+2 as 'MyResult'
END//
call proc(2);
```

```
+----------   +
| MyResult |
+----------   +
|       4      |
+----------   +
```

3)CREATE PROCEDURE proc_out (OUT var VARCHAR(100))

```
        -> BEGIN
        -> SET var='This return value';
        -> END//
```

```
mysql> DELIMITER ;
mysql> call proc_out(@op);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @op;
+------------------+
| @op                 |
+------------------+
| This return value |
+------------------+
1 row in set (0.00 sec)
```

То  есть ты при вызове определяешь переменную op, и передаешь ее в процедуру. Эта же переменная и возвращается. @-знак переменной.

4) INOUT

```
CREATE PROCEDURE pinout(INOUT p INT)
begin
```

```
        set @x=p*2;
        set p=1;
end//
set @a=5;
call pinout(@a);
select @a;
+------+
| @a   |
+------+
|        1 |
+------+
select @x;
+------+
| @x   |
+------+
|   10 |
+------+
```

То есть можно объявлять переменные в процедуре, и они будут доступны вне ее.

## ТРИГГЕРЫ.

Создадим простейший триггер на вставку.

```
mysql> CREATE TABLE tabl(id INT, value INT);
mysql> CREATE TRIGGER trig BEFORE INSERT ON tabl
        -> FOR EACH ROW SET @sum=@sum+NEW.value;
```

set @sum=0; - обязательно до вставки (инициализация знач тр.)

```
mysql> INSERT tabl VALUES(1,100);
```

```
mysql> INSERT tabl VALUES(2,40);

mysql> select @sum;

+------+
| @sum |
+------+
|  140 |
+------+
```

**Синтаксис создания триггера**
**CREATE**
[DEFINER = { имя_ пользователя | CURRENT_USER }]
**TRIGGER** имя_триггера время_триггера
событие_срабатывания_триггера
ON имя_таблицы FOR EACH ROW
выражение_выполняемое_при_срабатывании_триггера

Если с именем триггера и именем пользователя все понятно сразу, то о «времени триггера» и «событии» поговорим отдельно.

**время_триггера**

Определяет время свершения действия триггера. BEFORE означает, что триггер выполнится до завершения события срабатывания триггера, а AFTER означает, что после. Например, при вставке записей (см. пример выше) наш триггер срабатывал до фактической вставки записи и вычислял сумму. Такой вариант уместен при предварительном вычислении каких-то дополнительных полей в таблице или параллельной вставке в другую таблицу.

**событие_срабатывания_триггера**

Здесь все проще. Тут четко обозначается, при каком событии выполняется триггер.

- INSERT: т.е. при операциях вставки или аналогичных ей выражениях (INSERT, LOAD DATA, и REPLACE)
- UPDATE: когда сущность (строка) модифицирована
- DELETE: когда запись удаляется (запросы, содержащие выражения DELETE и/или REPLACE)

Пример логирования

```
CREATE TABLE test(id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY, content TEXT NOT NULL);


CREATE TABLE log(id int UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY, msg VARCHAR(255) NOT NULL, time TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP , row_id INT NOT NULL);


mysql> DELIMITER //
mysql> CREATE TRIGGER log_trig AFTER INSERT ON test FOR EACH ROW BEGIN INSERT INTO log SET msg='insert',row_id=NEW.id;
    -> END//


mysql> DELIMITER ;
```

```
mysql> INSERT INTO test VALUES('hello world');
ERROR 1136 (21S01): Column count doesn't match value count at row 1
mysql> INSERT INTO test(content) VALUES('hello world');
Query OK, 1 row affected (0.09 sec)

mysql> INSERT INTO test(content) VALUES('vova');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO test(content) VALUES('dasha');
Query OK, 1 row affected (0.04 sec)

mysql> SELECT * FROm log;
+----+--------+---------------------+--------+
| id | msg    | time                | row_id |
+----+--------+---------------------+--------+
|  1 | insert | 2013-12-17 13:42:40 |      1 |
|  2 | insert | 2013-12-17 13:42:45 |      2 |
|  3 | insert | 2013-12-17 13:42:49 |      3 |
+----+--------+---------------------+--------+
```

## ПРО ДВИЖКИ

**отсюда**

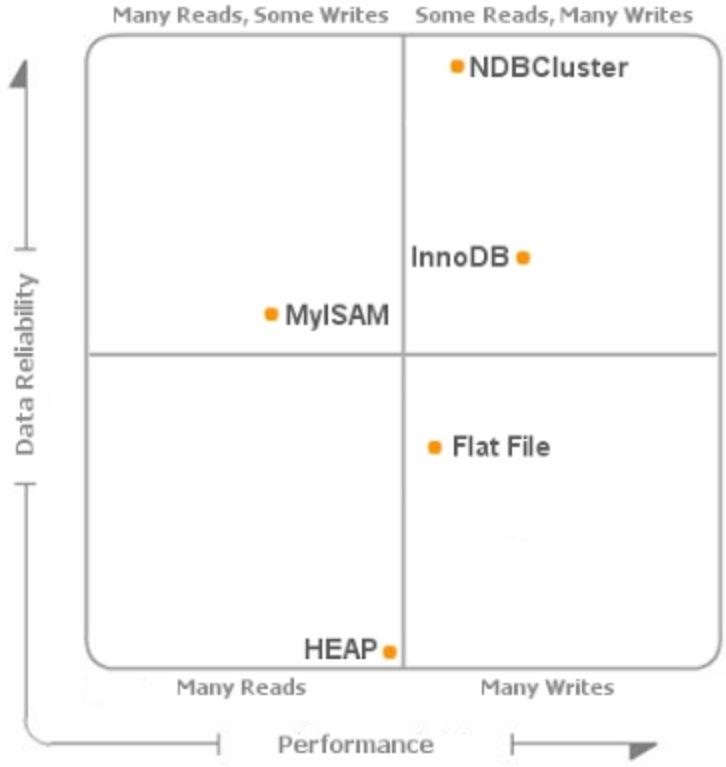
Одна из великолепных возможностей MySQL, отличная от бесплатности, широкой поддержки и быстроты, заключается в выборе различных движков хранения данных (storage engines) для различных таблиц.
MySQL предлагает 7 движков хранения данных, включая «example» — который позволяет Вам реализовать собственную библиотеку

хранения.

Что-же такого великолепного в обладании всеми этими вариантами?
Каждый движок хранения данных полностью различен и спроектирован для конкретных нужд приложений.
Не зацикливаясь на одном движке (как Oracle), Вы тем самым можете оптимизировать и выбирать лучший инструмент для своей работы.
Совет: Хорошо спроектированное приложение, активно использующее MySQL, должно использовать различные движки хранения данных для различных таблиц. Если Вы все еще завязли на MyISAM таблицах, может теперь стоит, что-то пересмотреть.

Обзор движков хранения данных MySQL
MyISAM: Движок по умолчанию. Не поддерживает транзакций, средняя надежность хранения данных. Превосходная производительность чтения данных для интенсивных приложений. Большинство веб сервисов и хранилищ данных активно используют MyISAM.
HEAP: Все в памяти. Очень быстрый поиск данных, однако все они хранятся только в памяти и будут потеряны при остановке сервера. Великолепно подходит для временных таблиц.
Archive: Используется для хранения больших объемов данных без индексов, занимая меньший размер.
Merge: Коллекция MyISAM таблиц логически объединенных вместе для единого представления.
InnoDB: Транзакционный тип движка, применяемый при интенсивных операциях записи, спасибо возможности блокировки уровня строк таблицы. Великолепная восстанавливаемость и высокая надежность хранения данных. Движок InnoDB был приобретен Oracle в 2005 году.
NDB: Кластерный движок — данные автоматически разделяются и реплицируются по различным машинам, именуемым — дата узлы. Применяется для приложений, которые требуют высокой производительности с наивысшей степенью доступности. NDB хорошо

работает на системах требующих высокой отдачи на операциях чтения. Для «тяжелых» приложений требующих активной записи в конкурирующей среде рассмотрите вариант с InnoDB.
Что-бы лучше понять уникальные характеристики каждого движка хранения данных, посмотрите на эту «магическую» диаграмму:




Примеры:
Ниже приведены несколько примеров использования наиболее подходящих движков хранения для различных задач:

- Поисковый движок — NDB кластер
- Логирование веб статистики — Обычные файлы для логирования с оффлайновым обработчиком и записью всей статистики в InnoDB таблицы
- Финансовые транзакции — InnoDB
- Сессионные данные — MyISAM или NDB кластер
- Локальные расчеты — HEAP
- Словари — MyISAM

Важные замечания по MyISAM таблицам:

- Таблицы могут быть повреждены. Ежедневно архивируйте Ваши данные или установите еще один MySQL сервер для выполнения репликаций.
- Включите авто-восстановление (auto-repair) в настройках Вашего сервера (my.cnf):
- *myisam-recover=backup,force*
- или рассмотрите возможность выполнения ежедневной автоматической проверки таблиц баз данных.
- Очень быстрое чтение данных (через SELECT)
- Конкурирующие записи полностью блокируют таблицы. Переключите все, что возможно, на оффлайн обработку записей сериями, что-бы не загружать движок сервера баз данных. (Оффлайн обработка — золотое правило, применимое для всех типов таблиц)

Важные замечания по HEAP/Memory таблицам:

Пока этот тип таблиц предлагает супер быстрый возврат данных, который работает только для небольших временных таблиц.
Если Вы загружаете слишком большие объемы данных в Memory таблицы, MySQL начинает свопировать информацию на диск и тем самым Вы теряете все преимущества Memory движка.

Важные замечания по InnoDB таблицам:

- Поддержка ACID транзакций. Встроенная отказоустойчивость данных, равная надежности 99.999%. Блокировка уровня строк (сравните с полной блокировкой всей таблицы в MyISAM) означает обеспечение быстрой записи конкурирующих операций.
- Выполнение «SELECT Count(*) FROM table» без индексов выполняется в InnoDB очень медленно и требует сканирования всей таблицы. (Для MyISAM эта операция ничего не стоит, потому что он хранит внешние записи счетчиков для каждой таблицы).

- Если Вам часто необходима операция «SELECT COUNT(*)» на таблицах InnoDB, создайте MySQL триггер на вставку/удаление, который будет увеличивать/уменьшать счетчик, когда данные добавляются или удаляются из таблицы.
- Резервирование (бакапирование)
- Простое архивирование всех файлов таблиц для InnoDB невозможно.
- MySQLDump резервирует InnoDB очень медленно. (Если Вы настаиваете на таком резервировании, включите флаг: —opt —compress)
- Быстрое жизнеспособное резервирование, которое так-же может быть использовано как новая «ведомая» (slave) машина, это InnoDB Hot Backup.
- Восстановление
- В InnoDB встроена поддержка восстановления, которая работает в 99% случаев автоматически. Никогда не трогайте .frm или .ibd файлы в надежде «помочь» восстановлению базы данных. Если встроенное восстановление не сработало, переключайтесь на «ведомый» сервер и восстанавливайте основной из архивов.
- LOAD DATA INFILE в InnoDB работает очень медленно. Для операций LOAD DATA присмотритесь к использованию MyISAM таблиц.
- InnoDB меньше, чем MyISAM, прощает выполнение запросов построенных не на индексах. InnoDB отправит Вас в «школу», что-бы быть уверенным, что каждый запрос или обновление будет запущено на индексах. Выполните непроиндексированный запрос и Вы поплатитесь за это временем исполнения.
- Никогда не изменяйте my.cnf InnoDB лог файл, когда запущен сервер баз данных. Вы разрушите последовательный лог-номер (log sequence number) оставшись без возможность восстановления.
- Для увеличения производительности InnoDB, присмотритесь к

использованию следующих настроек (my.cnf):

- 
- *innodb_open_files = 500*
- *innodb_file_per_table*
- *innodb_buffer_pool_size = 250M*
- *innodb_flush_log_at_trx_commit = 2*
- *innodb_thread_concurrency =8*
- *innodb_lock_wait_timeout = 500*
- *interactive_timeout = 20*
- *back_log = 75*
- *table_cache = 300*
- *thread_cache = 32*
- *thread_concurrency = 8*
- *wait_timeout = 30*
- *connect_timeout = 10*
- 

Расширяемость

Каждое успешное веб приложение, в конце концов, перерастает возможности сервера баз данных размещающегося на одной машине. В этом случае, обычно, Вы имеете две возможности — Репликации или Кластер NDB. Выбор зависит от требований Вашего приложения. Для активно-читающей (read-heavy) среды, используйте NDB кластер или установите репликации для n MyISAM ведомых read-only машин. Для активно-пишущей (write-heavy) среды, InnoDB с активно/пассивными репликациями будет лучшим типовым решением. Но Вы можете поэкспериментировать с NDB кластером. NDB кластер обычно медленнее чем InnoDB в операциях с активной записью, но он предлагает наивысший уровень доступности.

# Уровни транзакции

Есть таблица
CREATE TABLE test (id INT, value VARCHAR(255)) ENGINE=InnoDB;

Что по Вашему покажет этот запрос?
START TRANSACTION;
INSERT INTO test(id, value) VALUES (1, 'test'), (2, 'test 2');
SELECT * FROM test;
COMMIT;
SELECT * FROM test;

А что покажет простейший SELECT во время выполнения текущей транзакции? Не ясно. Вот и придумали такие правила.

Первый **READ UNCOMMITTED**
Рассмотрим транзакцию выше. После INSERT данные сразу-же станут доступны для чтения. Тоесть еще до вызова COMMIT вне транзакции можно получить только что добавленные данные. В английской литературе это называется dirty read («грязное чтение»). Этот уровень редко используется на практике, да вообще редко кто меняет эти самые уровни.

Второй **READ COMMTITED**
В данном случае прочитать данные возможно только после вызова COMMIT. При чем внутри транзакции данные тоже будут еще не доступны.

Если рассмотреть транзакцию выше, то первый SELECT ничего не вернет, т.к. таблица у нас еще пустая и транзакция не подтверждена.

Третий **REPEATABLE READ**
Этот уровень используется по умолчанию в MySQL. Отличается от второго тем, что вновь добавленные данные уже будут доступны внутри транзакции, но не будут доступны до подтверждения извне. Здесь может возникнуть теоретическая проблема «фантомного чтения». Когда внутри одной транзакции происходит чтение данных, другая транзакция в этот момент вставляет новые данные, а первая транзакция снова читает те-же самые данные.

И последний **SERIALIZABLE**
На данном уровне MySQL блокирует каждую строку над которой происходит какое либо действие, это исключает появление проблемы «фантомов». На самом деле смысла использовать этот уровень нет, т.к. InnoDB и менее популярный Falcon решают эту проблему.

Увидеть текущий уровень изоляции
SHOW VARIABLES LIKE '%tx_isolation%';

Установить
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

Это всего лишь попытка вольного перевода самой обычной документации.
Спасибо.

Задачка:

```
CREATE TABLE one(id INT, value VARCHAR(12)) ENGINE = MyISAM;
CREATE TABLE two(id INT, value VARCHAR(12)) ENGINE = InnoDB;

START TRANSACTION;
INSERT INTO one (id, value) VALUES (1, 'test');
INSERT INTO two (id, value) VALUES (1, 'test');
ROLLBACK;
SELECT * FROM one;
SELECT * FROM two;
```

Что будет в первой таблице, а что во второй?

## JOIN

**JOIN** — оператор языка SQL, который является реализацией операции соединения реляционной алгебры. Входит в раздел FROM операторов SELECT, UPDATE или DELETE.
Операция соединения, как и другие бинарные операции, предназначена для обеспечения выборки данных из двух таблиц и включения этих данных в один результирующий набор. Отличительной особенностью операции соединения является следующее:

- в схему таблицы-результата входят столбцы обеих исходных таблиц (таблиц-операндов), то есть схема результата является «сцеплением» схем операндов;
- каждая строка таблицы-результата является «сцеплением» строки из одной таблицы-операнда со строкой второй таблицы-операнда.

Определение того, какие именно исходные строки войдут в результат и в каких сочетаниях, зависит от типа операции соединения и от явно заданного *условия соединения*. Условие соединения, то есть условие

сопоставления строк исходных таблиц друг с другом, представляет собой логическое выражение (предикат).
При необходимости соединения не двух, а нескольких таблиц, операция соединения применяется несколько раз (последовательно).

## Описание оператора

```
SELECT
  FIELD [,... n]
FROM
  Table1
  {INNER | {LEFT | RIGHT | FULL} OUTER | CROSS } JOIN
  Table2
    ON <condition>
```

В большинстве СУБД при указании слов LEFT, RIGHT, FULL слово OUTER можно опустить. Слово INNER также в большинстве СУБД можно опустить.
В общем случае СУБД при выполнении соединения проверяет условие (предикат) *condition*. Для CROSS JOIN условие не указывается.
Для перекрёстного соединения (декартова произведения) CROSS JOIN в некоторых реализациях SQL используется оператор «запятая» (,):

```
SELECT
  FIELD [,... n]
FROM
  Table1,
  Table2
```

## Виды оператора JOIN

Для дальнейших пояснений будут использоваться следующие таблицы:

| Id | Name |
|---|---|
| 1 | Москва |
| 2 | Санкт-Петербург |
| 3 | Казань |

| Name | CityId |
|---|---|
| Андрей | 1 |
| Леонид | 2 |
| Сергей | 1 |
| Григорий | 4 |

#### INNER JOIN

Оператор *внутреннего соединения* INNER JOIN соединяет две таблицы. Порядок таблиц для оператора неважен, поскольку оператор является симметричным.
*Заголовок* таблицы-результата является объединением (конкатенацией) заголовков соединяемых таблиц.
*Тело* результата логически формируется следующим образом. Каждая строка одной таблицы сопоставляется с каждой строкой второй таблицы, после чего для полученной «соединённой» строки проверяется условие соединения (вычисляется предикат соединения). Если условие истинно, в таблицу-результат добавляется соответствующая «соединённая» строка.

Описанный алгоритм действий является строго логическим, то есть он лишь объясняет результат, который должен получиться при выполнении операции, но не предписывает, чтобы конкретная СУБД выполняла соединение именно указанным образом. Существует множество способов реализации операции соединения, например соединение вложенными циклами (англ. *inner loops join*), соединение хэшированием (англ. *hash join*), соединение слиянием (англ. *merge join*). Единственное требование состоит в том, чтобы любая реализация логически давала такой же результат, как при применении описанного алгоритма.

```
SELECT *
FROM
  Person
  INNER JOIN
  City
    ON Person.CityId = City.Id
```

Результат:

| Person.Name | Person.CityId | City.Id | City.Name |
|---|---|---|---|
| Андрей | 1 | 1 | Москва |
| Леонид | 2 | 2 | Санкт-Петербург |
| Сергей | 1 | 1 | Москва |

**OUTER JOIN**

Соединение двух таблиц, в результат которого в обязательном порядке входят строки либо одной, либо обеих таблиц.

**LEFT OUTER JOIN**

Оператор *левого внешнего соединения* LEFT OUTER JOIN соединяет две таблицы. Порядок таблиц для оператора важен,

поскольку оператор не является симметричным.
*Заголовок* таблицы-результата является объединением (конкатенацией) заголовков соединяемых таблиц.
*Тело* результата логически формируется следующим образом. Пусть выполняется соединение левой и правой таблиц по предикату (условию) *p*.

1. В результат включается внутреннее соединение (INNER JOIN) левой и правой таблиц по предикату *p*.
2. Затем в результат добавляются те записи левой таблицы, которые не вошли во внутреннее соединение на шаге 1. Для таких записей поля, соответствующие правой таблице, заполняются значениями *NULL*.

```
SELECT *
FROM
  Person
  LEFT OUTER JOIN
  City
    ON Person.CityId = City.Id
```

Результат:

| Person.Name | Person.CityId | City.Id | City.Name |
|---|---|---|---|
| Андрей | 1 | 1 | Москва |
| Леонид | 2 | 2 | Санкт-Петербург |
| Сергей | 1 | 1 | Москва |
| Григорий | 4 | NULL | NULL |

**RIGHT OUTER JOIN**

Оператор *правого внешнего соединения* RIGHT OUTER JOIN соединяет две таблицы. Порядок таблиц для оператора важен, поскольку оператор не является симметричным.

*Заголовок* таблицы-результата является объединением (конкатенацией) заголовков соединяемых таблиц.
*Тело* результата логически формируется следующим образом. Пусть выполняется соединение левой и правой таблиц по предикату (условию) *p*.

1. В результат включается внутреннее соединение (INNER JOIN) левой и правой таблиц по предикату *p*.
2. Затем в результат добавляются те записи правой таблицы, которые не вошли во внутреннее соединение на шаге 1. Для таких записей поля, соответствующие левой таблице, заполняются значениями *NULL*.

```
SELECT *
FROM
  Person
  RIGHT OUTER JOIN
  City
    ON Person.CityId = City.Id
```

Результат:

| Person.Name | Person.CityId | City.Id | City.Name |
|---|---|---|---|
| Андрей | 1 | 1 | Москва |
| Сергей | 1 | 1 | Москва |
| Леонид | 2 | 2 | Санкт-Петербург |
| NULL | NULL | 3 | Казань |

**FULL OUTER JOIN**

Оператор *полного внешнего соединения* FULL OUTER JOIN соединяет две таблицы. Порядок таблиц для оператора неважен, поскольку оператор является симметричным.
*Заголовок* таблицы-результата является объединением

(конкатенацией) заголовков соединяемых таблиц.
*Тело* результата логически формируется следующим образом. Пусть выполняется соединение первой и второй таблиц по предикату (условию) *p*. Слова «первой» и «второй» здесь не обозначают порядок в записи (который неважен), а используются лишь для различения таблиц.

1. В результат включается внутреннее соединение (INNER JOIN) первой и второй таблиц по предикату *p*.
2. В результат добавляются те записи первой таблицы, которые не вошли во внутреннее соединение на шаге 1. Для таких записей поля, соответствующие второй таблице, заполняются значениями *NULL*.
3. В результат добавляются те записи второй таблицы, которые не вошли во внутреннее соединение на шаге 1. Для таких записей поля, соответствующие первой таблице, заполняются значениями *NULL*.

```
SELECT *
FROM
  Person
  FULL OUTER JOIN
  City
    ON Person.CityId = City.Id
```

Результат:

| Person.Name | Person.CityId | City.Id | City.Name |
|---|---|---|---|
| Андрей | 1 | 1 | Москва |
| Сергей | 1 | 1 | Москва |
| Леонид | 2 | 2 | Санкт-Петербург |
| NULL | NULL | 3 | Казань |

| | | | |
|---|---|---|---|
| Григорий | 4 | NULL | NULL |

**CROSS JOIN**

Оператор *перекрёстного соединения*, или *декартова произведения* CROSS JOIN соединяет две таблицы. Порядок таблиц для оператора неважен, поскольку оператор является симметричным.
*Заголовок* таблицы-результата является объединением (конкатенацией) заголовков соединяемых таблиц.
*Тело* результата логически формируется следующим образом. Каждая строка одной таблицы соединяется с каждой строкой второй таблицы, давая тем самым в результате все возможные сочетания строк двух таблиц.

```
SELECT *
FROM
  Person
  CROSS JOIN
  City
```

или

```
SELECT *
FROM
  Person,
  City
```

Результат:

| Person.Name | Person.CityId | City.Id | City.Name |
|---|---|---|---|
| Андрей | 1 | 1 | Москва |
| Андрей | 1 | 2 | Санкт-Петербург |
| Андрей | 1 | 3 | Казань |

| | | | |
|---|---|---|---|
| Леонид | 2 | 1 | Москва |
| Леонид | 2 | 2 | Санкт-Петербург |
| Леонид | 2 | 3 | Казань |
| Сергей | 1 | 1 | Москва |
| Сергей | 1 | 2 | Санкт-Петербург |
| Сергей | 1 | 3 | Казань |
| Григорий | 4 | 1 | Москва |
| Григорий | 4 | 2 | Санкт-Петербург |
| Григорий | 4 | 3 | Казань |

Если в предложении WHERE добавить условие соединения, то есть ограничения на сочетания кортежей, то результат эквивалентен операции INNER JOIN с таким же условием:

```
SELECT *
FROM
  Person,
  City
WHERE
  Person.CityId = City.Id
```

Таким образом, CROSS JOIN + WHERE(*предикат*) и INNER JOIN(*предикат*) синтаксически являются альтернативными формами записи одной и той же логической операции внутреннего соединения. Синтаксис CROSS JOIN + WHERE для операции соединения называют устаревшим,[1][2] поскольку его не рекомендует стандарт SQL ANSI.