Общие подходы

Мы посмотрим на то, как подойти правильно к оптимизации СУБД, а также посмотрим на некоторые практические подходы к этому делу. Первая вещь, о которой нужно знать, что производительность СУБД сама по себе на самом деле не важна. На самом деле вашим пользователям, вашему боссу важна, прежде всего, производительность вашего приложения, которое использует СУБД, а может использовать какие-то другие системы. И это то, на чем имеет смысл фокусироваться, потому что если вы смотрите на проблему с точки зрения приложения, то вам открываются некоторые другие, более широкие подходы.

Когда мы говорим о производительности, о чем мы думаем? Прежде всего мы думаем о времени отклика. Почему? Потому что если вы посмотрите на вашего эгоистичного пользователя, то это то, о чем он думает прежде всего. Т.е. если я захожу на веб-сайт или работаю с приложением, и это приложение мне отвечает быстро, то это все, что меня волнует.

Синтаксис оператора `EXPLAIN` (получение информации о `SELECT`)

Использовать оператор EXPLAIN просто. Его необходимо добавлять в запросы перед оператором SELECT.

Вывод может не выглядеть точь-в-точь так, тем не менее, в нем будут содержаться те же 10 столбцов. Что же это за возвращаемые столбцы?

- **id** – порядковый номер для каждого SELECT'а внутри запроса (когда имеется несколько подзапросов)
- **select_type** – тип запроса SELECT.
    - **SIMPLE** — Простой запрос SELECT без подзапросов или UNION'ов
    - **PRIMARY** – данный SELECT – самый внешний запрос в JOIN'е
    - **DERIVED** – данный SELECT является частью подзапроса внутри FROM
    - **SUBQUERY** – первый SELECT в подзапросе
    - **DEPENDENT SUBQUERY** – подзапрос, который зависит от внешнего запроса
    - **UNCACHABLE SUBQUERY** – не кешируемый подзапрос (существуют определенные условия для того, чтобы запрос кешировался)
    - **UNION** – второй или последующий SELECT в UNION'е
    - **DEPENDENT UNION** – второй или последующий SELECT в UNION'е, зависимый от внешнего запроса
    - **UNION RESULT** – результат UNION'а
- **Table** – таблица, к которой относится выводимая строка
- **Type** — указывает на то, как MySQL связывает используемые таблицы. Это одно из наиболее полезных полей в выводе потому, что может сообщать об отсутствующих индексах или почему написанный запрос должен быть пересмотрен и переписан.
  Возможные значения:
    - **System** – таблица имеет только одну строку
    - **Const** – таблица имеет только одну соответствующую строку, которая проиндексирована. Это наиболее быстрый тип соединения потому, что таблица читается только один раз и значение строки может восприниматься при дальнейших соединениях как константа.

    - **Eq_ref** – все части индекса используются для связывания. Используемые индексы: PRIMARY KEY или UNIQUE NOT NULL. Это еще один наилучший возможный тип связывания.
    - **Ref** – все соответствующие строки индексного столбца считываются для каждой комбинации строк из предыдущей таблицы. Этот тип соединения для индексированных столбцов выглядит как использование операторов = или < = >
    - **Fulltext** – соединение использует полнотекстовый индекс таблицы
    - **Ref_or_null** – то же самое, что и ref, но также содержит строки со значением null для столбца
    - **Index_merge** – соединение использует список индексов для получения результирующего набора. Столбец key вывода команды EXPLAIN будет содержать список использованных индексов.
    - **Unique_subquery** – подзапрос IN возвращает только один результат из таблицы и использует первичный ключ.
    - **Index_subquery** – тоже, что и предыдущий, но возвращает более одного результата.
    - **Range** – индекс, использованный для нахождения соответствующей строки в определенном диапазоне, обычно, когда ключевой столбец сравнивается с константой, используя операторы вроде: BETWEEN, IN, >, >=, etc.
    - **Index** – сканируется все дерево индексов для нахождения соответствующих строк.
    - **All** – Для нахождения соответствующих строк используются сканирование всей таблицы. Это наихудший тип соединения и обычно указывает на отсутствие подходящих индексов в таблице.
- **Possible_keys** – показывает индексы, которые могут быть использованы для нахождения строк в таблице. На практике они могут использоваться, а могут и не использоваться. Фактически, этот столбец может сослужить добрую службу в деле оптимизации запросов, т.к значение NULL указывает на то, что не найдено ни одного подходящего индекса .
- **Key**– указывает на использованный индекс. Этот столбец может содержать индекс, не указанный в столбце possible_keys. В процессе соединения таблиц оптимизатор ищет наилучшие варианты и может найти ключи, которые не отображены в possible_keys, но являются более оптимальными для использования.
- **Key_len** – длина индекса, которую оптимизатор MySQL выбрал для использования. Например, значение key_len, равное 4, означает, что памяти требуется для хранения 4 знаков. На эту тему вот ссылка
- **Ref** – указываются столбцы или константы, которые сравниваются с индексом, указанным в поле key. MySQL выберет либо значение константы для сравнения, либо само поле, основываясь на плане выполнения запроса.
- **Rows** – отображает число записей, обработанных для получения выходных данных. Это еще одно очень важное поле, которое дает повод оптимизировать запросы, особенно те, которые используют JOIN'ы и подзапросы.
- **Extra** – содержит дополнительную информацию, относящуюся к плану выполнения запроса. Такие значения как "Using temporary", "Using filesort" и т.д могут быть индикатором проблемного запроса. С полным списком возможных значений вы можете ознакомиться здесь

После EXPLAIN в запросе вы можете использовать ключевое слово EXTENDED и MySQL покажет вам дополнительную информацию о том, как выполняется запрос. Чтобы увидеть эту информацию, вам нужно сразу после запроса с EXTENDED выполнить запрос SHOW WARNINGS. Наиболее полезно смотреть эту информацию о запросе, который выполнялся после каких-либо изменений сделанных оптимизатором запросов.

explain SELECT student.id, student.last_name, exam_result.result FROM student inner join exam_result on exam_result.student_id=student.id;

SIMPLE	student	index	PRIMARY,id	students_name	62		28	Using index
SIMPLE	exam_result	ref	exam_result_student_fk_idx	exam_result_student_fk_idx	4	students.student.id	1	

explain SELECT count(*),exam_result.result FROM student left join exam_result on exam_result.student_id=student.id where hostel_live=1 group by exam_result.result;

1	SIMPLE	student	ALL				28	Using where; Using temporary; Using filesort
1	SIMPLE	exam_result	ref	exam_result_student_fk_idx	exam_result_student_fk_idx	4	students.student.id	1

explain SELECT count(*),teacher.last_name,training_course.name,student_result.result FROM teacher left join training_course on training_course.teacher_id=teacher_id=teacher.id left join student_result on student_result.training_course_id=training_course.id group by teacher.id, training_course.id

1	SIMPLE	teacher	ALL				1	Using temporary; Using filesort
1	SIMPLE	training_course	ALL				15	
1	SIMPLE	student_result	ref	student_result__idx	student_result__idx	4	students.training_course.id	7

explain SELECT count(*),techer.last_name,training_course.name,student_result.result FROM techer left join training_course on training_course.teacher_id=techer.id left join student_result on student_result.training_course_id=training_course.id where student_result.result > 3 group by techer.id, training_course.id

1	SIMPLE	training_course	ALL	PRIMARY,id,teacher_fk_idx			15	Using temporary; Using filesort
1	SIMPLE	techer	eq_ref	PRIMARY,id	PRIMARY	4	students.training_course.teacher_id	1
1	SIMPLE	student_result	ref	student_result__idx	student_result__idx	4	students.training_course.id	7 Using where

Postgre

Структура плана запроса представляет собой дерево *узлов плана*. Узлы на нижнем уровне дерева — это узлы сканирования, которые возвращают необработанные данные таблицы. Разным типам доступа к таблице соответствуют разные узлы: последовательное сканирование, сканирование индекса и сканирование битовой карты. Источниками строк могут быть не только таблицы, но и например, предложения `VALUES` и функции, возвращающие множества во `FROM`, и они представляются отдельными типами узлов сканирования.

Если запрос требует объединения, агрегатных вычислений, сортировки или других операций с исходными строками, над узлами сканирования появляются узлы, обозначающие эти операции. И так как обычно операции могут выполняться разными способами, на этом уровне тоже могут быть узлы разных типов. В выводе команды `EXPLAIN` для каждого узла в дереве плана отводится одна строка, где показывается базовый тип узла плюс оценка стоимости выполнения данного узла, которую сделал для него планировщик.

Если для узла выводятся дополнительные свойства, в вывод могут добавляться дополнительные строки, с отступом от основной информации узла. В самой первой строке (основной строке самого верхнего узла) выводится общая стоимость выполнения для всего плана; именно это значение планировщик старается минимизировать.

explain SELECT student.id, student.last_name, exam_result.result FROM student inner join exam_result on exam_result.student_id=student.id;

	QUERY PLAN text
1	Hash Join (cost=20.80..37.00 rows=490 width=64)
2	Hash Cond: (exam_result.student_id = student.id)
3	-> Seq Scan on exam_result (cost=0.00..14.90 rows=490 width=6)
4	-> Hash (cost=14.80..14.80 rows=480 width=62)
5	-> Seq Scan on student (cost=0.00..14.80 rows=480 width=62)

Числа, перечисленные в скобках (слева направо), имеют следующий смысл:

- Приблизительная стоимость запуска. Это время, которое проходит, прежде чем начнётся этап вывода данных, например для сортирующего узла это время сортировки.
- Приблизительная общая стоимость. Она вычисляется в предположении, что узел плана выполняется до конца, то есть возвращает все доступные строки. На практике родительский узел может досрочно прекратить чтение строк дочернего (см. приведённый ниже пример с `LIMIT`).
- Ожидаемое число строк, которое должен вывести этот узел плана. При этом так же предполагается, что узел выполняется до конца.

- Ожидаемый средний размер строк, выводимых этим узлом плана (в байтах).

  Стоимость может измеряться в произвольных единицах, определяемых параметрами планировщика Традиционно единицей стоимости считается операция чтения страницы с диска; то есть seq_page_cost обычно равен `1.0`, а другие параметры задаётся относительно него. Примеры в этом разделе выполняются со стандартными параметрами стоимости.

  Важно понимать, что стоимость узла верхнего уровня включает стоимость всех его потомков. Также важно осознавать, что эта стоимость отражает только те факторы, которые учитывает планировщик. В частности, она не зависит от времени, необходимого для передачи результирующих строк клиенту, хотя оно может составлять значительную часть общего времени выполнения запроса. Тем не менее планировщик игнорирует эту величину, так как он всё равно не сможет изменить её, выбрав другой план. (Мы верим в то, что любой правильный план запроса выдаёт один и тот же набор строк.)

  Значение `rows` здесь имеет особенность — оно выражает не число строк, обработанных или просканированных узлом плана, а число строк, выданных этим узлом. Часто оно окажется меньше числа просканированных строк в результате применённой к узлу фильтрации по условиям `WHERE`. В идеале, на верхнем уровне это значение будет приблизительно равно числу строк, которое фактически возвращает, изменяет или удаляет запрос.

Точность оценок планировщика можно проверить, используя команду `EXPLAIN` с параметром `ANALYZE`. С этим параметром `EXPLAIN` на самом деле выполняет запрос, а затем выводит фактическое число строк и время выполнения, накопленное в каждом узле плана, вместе с теми же оценками, что выдаёт обычная команда `EXPLAIN`.

Время выполнения, измеренное командой `EXPLAIN ANALYZE`, может значительно отличаться от времени выполнения того же запроса в обычном режиме. Тому есть две основных причины. Во-первых, так как при анализе никакие строки результата не передаются клиенту, время ввода/вывода и передачи по сети не учитывается. Во-вторых, может быть существенной дополнительная нагрузка, связанная с функциями измерений `EXPLAIN ANALYZE`, особенно в системах, где вызов `gettimeofday()` выполняется медленно. Для измерения этой нагрузки вы можете воспользоваться утилитой pg_test_timing.

Результаты `EXPLAIN` не следует распространять на ситуации, значительно отличающиеся от тех, в которых вы проводите тестирование. В частности, не следует полагать, что выводы, полученные для игрушечной таблицы, будут применимы и для настоящих больших таблиц. Оценки стоимости нелинейны и планировщик может выбирать разные планы в зависимости от размера таблицы. Например, в крайнем случае вся таблица может уместиться в одну страницу диска, и тогда вы почти наверняка получите план последовательного сканирования, независимо от того, есть у неё и индексы или нет. Планировщик понимает, что для обработки таблицы ему в любом случае потребуется прочитать одну страницу, так что нет никакого смысла обращаться к ещё одной странице за индексом.

Панировщик запросов должен оценить число строк, возвращаемых запросов, чтобы сделать правильный выбор в отношении плана запроса. Для большей эффективности `периодически нужно` обновлять статистику с помощью команды `VACUUM`, `ANALYZE` и несколько команд DDL.

Оконные функции

window function выполняет вычисления над списком строк в таблице, которые как-то относятся к текущей строке. Это сравнимо с типом вычислений, которые могут быть выполнены с помощью какой-либо агрегатной функции. Но в отличие от обычных агрегатных функций, использование оконной функции не заставляет строки группироваться в одну; строки сохраняют свои отдельные значения. Другими словами, оконная функция позволяет получить доступ более чем только к текущей строке результата запроса.

```
SELECT depname, empno, salary, date, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
  depname  | empno | salary | date     |          avg
-----------+-------+--------+----------+-----------------------
 develop   |    11 |   5200 |10.01.2019| 5020.0000000000000000
 develop   |     7 |   4200 |10.02.2019| 5020.0000000000000000
 develop   |     9 |   4500 |10.03.2019| 5020.0000000000000000
 develop   |     8 |   6000 |10.04.2019| 5020.0000000000000000
 develop   |    10 |   5200 |10.05.2019| 5020.0000000000000000
 personnel |     5 |   3500 |11.01.2019| 3700.0000000000000000
 personnel |     2 |   3900 |12.01.2019| 3700.0000000000000000
 sales     |     3 |   4800 |03.01.2019| 4866.6666666666666667
 sales     |     1 |   5000 |04.01.2019| 4866.6666666666666667
 sales     |     4 |   4800 |05.01.2019| 4866.6666666666666667
(10 rows)
```

Первые четыре колонки берутся непосредственно из таблицы `empsalary` и в результат запроса попадает каждая строка в этой таблице. Последняя колонка является средним значением, подсчитанным для всех строк таблицы, которые имеют то самое значение в колонке `depname` как и в текущей строке. (Фактически это такая же как и обычная агрегатная функция `avg`, но предложение `OVER` заставляет её работать как оконную функцию и производить подсчёт соответствующего списка строк.)

Любой вызов оконной функции всегда содержит предложение `OVER`, за которым сразу следует имя оконной функции и аргумент(ы). В этом и заключается её синтаксическое отличие от обычной функции или агрегатной функции. Предложение `OVER` точно определяет какие строки в запросе разбиваются для обработки оконной функцией. Список `PARTITION BY` внутри `OVER` задаёт деление строк на группы или разбиения, которые разделяют те же самые значения выражения(й) `PARTITION BY`. Для каждой строки, оконной функция обсчитывает только строки, которые попадают в то же самое разбиение, что и текущая строка.

Если запрос содержит оконные, эти функции вычисляются после каждой группировки, агрегатных выражений и фильтрации `HAVING`. Другими словами, если в запросе есть агрегатные функции, предложения `GROUP BY` или `HAVING`, оконные функции видят не исходные строки, полученные из `FROM`/`WHERE`, а сгруппированные.

Когда используются несколько оконных функций, все оконные функции, имеющие в своих определениях синтаксически равнозначные предложения `PARTITION BY` и `ORDER BY`, гарантированно обрабатывают данные за один проход. Таким образом, они увидят один порядок

сортировки, даже если `ORDER BY` не определяет порядок однозначно. Однако относительно функций с разными формулировками `PARTITION BY` и `ORDER BY` никаких гарантий не даётся. (В таких случаях между проходами вычислений оконных функций обычно требуется дополнительный этап сортировки и эта сортировка может не сохранять порядок строк, равнозначный с точки зрения `ORDER BY`.)

В настоящее время оконные функции всегда требуют предварительно отсортированных данных, так что результат запроса будет отсортирован согласно тому или иному предложению `PARTITION BY`/`ORDER BY` оконных функций. Однако полагаться на это не следует. Если вы хотите, чтобы результаты сортировались определённым образом, явно добавьте предложение `ORDER BY` на верхнем уровне запроса.

Приведение типов

Приведение типа определяет преобразование данных из одного типа в другой. Postgres Pro воспринимает две равносильные записи приведения типов:

```
CAST ( выражение AS тип )
выражение::тип
```

Оптимальная настройка Mysql

innodb_buffer_pool_size

Если Вы используете только InnoDB таблицы, устанавливайте это значение максимально возможным для Вашей системы. Буфер InnoDB кеширует и данные и индексы. Поэтому значение этого ключа стоит устанавливать в *70%...80%* всей доступной памяти.

innodb_log_file_size

Эта опция влияет на скорость записи. Она устанавливает размер лога операций (так операции сначала записываются в лог, а потом применяются к данным на диске). Чем больше этот лог, тем быстрее будут работать записи (т.к. их поместится больше в файл лога). Файлов всегда два, а их размер одинаковый. Значением параметра задается размер одного файла. Стоит понимать, что увеличение этого параметра увеличит и время восстановления системы при сбоях. Это происходит потому, что при запуске системы все данные из логов будет накатываться на данные. Однако с каждой новой версией, производительность этого процесса растет. Подумайте над использованием реплик для обеспечения доступности, чтобы не зависеть от времени восстановления базы данных.

innodb_log_buffer_size

Это размер буфера транзакций, которые не были еще закомичены. Значение этого параметра стоит менять в случаях, если вы используете большие поля вроде BLOB или TEXT.

innodb_file_per_table

Если включить эту опцию, Innodb будет сохранять данные всех таблиц в отдельных файлах (вместо одного файла по умолчанию). Прироста в производительности не будет, однако есть ряд преимуществ:

- При удалении таблиц, диск будет освобождаться. По умолчанию общий файл данных может только расширяться, но не уменьшаться.
- Использование компрессионного формата таблиц потребует включить этот параметр.

innodb_flush_log_at_trx_commit

Изменение этого параметра может повысить пропускную способность записи данных в базу в сотни раз. Он определяет, будет ли Mysql сбрасывать **каждую операцию** на диск (в файл лога).

Тут следует руководствоваться такой логикой:

- *innodb_flush_log_at_trx_commit = 1* для случаев, когда сохранность данных — это приоритет номер один.
- *innodb_flush_log_at_trx_commit = 2* для случаев, когда небольшая потеря данных не критична (например, вы используете дублирование и сможете восстановить небольшую потерю). В этом случае транзакции будут сбрасываться в лог на диск только раз в секунду.

query_cache_size

Значение этого параметра определяет сколько памяти стоит использовать под кеш запросов. Самый правильный подход — не полагаться на этот механизм. На практике он работает очень неэффективно. Так, весь кеш запросов для определенной таблицы сбрасывается всякий раз, когда в таблицу вносится хотя бы одно изменение. Это может привести к тому, что включение кеширования даже замедлит базу данных

max_connections

Не следует изменять значение этого параметра на старте. Однако, если вы получаете ошибки *"Too many connections"*, эту опцию стоит поднимать. Она определяет максимальное количество одновременных соединений с базой данных

Тюнинг базы Postgres

listen_addresses

По умолчанию Postgres принимает соединения только с локальных служб, т.к. слушает интерфейс *localhost*. Если Вы планируете сетевую подсистему, состоящую из более, чем одного компьютера (Ваш сервер баз данных будет находится на отдельном компьютере), Вам потребуется поменять этот параметр

max_connections

Этот параметр определяет максимальное количество одновременных соединений, которые будет обслуживать сервер. В принципе, это число должно определяться исходя из требований к системе. Этот параметр в большей степени влияет на использование ресурсов. Если Вы только стартуете, устанавливайте это значение небольшим (16...32), постепенно увеличивая его (по мере необходимости — такой мерой будет получение ошибок от postgres "too many clients").

Учтите! На поддержку каждого активного клиента, postgres тратит немалое количество ресурсов, и если Вам необходимо добиться производительности в несколько тысяч активных соединений, то стоит использовать менеджеры соединений, например: Pgpool.

shared_buffers

Этот параметр определяет, сколько памяти будет выделяться postgres для кеширования данных. В стандартной поставке значение этого параметра мизерное — для обеспечения совместимости. В практических условиях это значение следует установить в 15..25% от всей доступной оперативной памяти.

effective_cache_size

Этот параметр помогает планировщику postgres определить количество доступной памяти для дискового кеширования. На основе того, доступна память или нет, планировщик будет делать выбор между использованием индексов и использованием сканирования таблицы. Это значение следует устанавливать в 50%...75% всей доступной оперативной памяти, в зависимости от того, сколько памяти доступно для системного кеша. **Еще раз** — этот параметр не влияет на выделяемые ресурсы — это оценочная информация для планировщика.

checkpoint_segments

На эту настройку стоит обратить внимание, если у Вас происходит немалое количество записей в БД (для высоконагруженных систем это нормальная ситуация). Postgres записывает данные в базу данных порциями (WALL сегменты) — каждая размером в 16Mb. После записи определенного количества таких порций (определяется параметром *checkpoint_segments*) происходит чекпойнт. Чекпойнт — это набор операций, которые выполняет postgres для гарантии того, что все изменения были записаны в файлы данных (следовательно при сбое, восстановление происходит по последнему чекпойнту). Выполнение чекпоинтов каждые 16Мб может быть весьма ресурсоемким, поэтому это значение следует увеличить хотя бы до *10*.

Для случаев с большим количеством записей, стоит увеличивать это значение в рамках от 32 до 256.

work_mem

Важный параметр для запросов, использующих всевозможные сложные выборки и сортировки. Увеличение его позволяет выполнять эти операции в оперативной памяти, что гораздо более эффективно, чем на диске (еще бы). **Будьте внимательны!** Этот параметр указывает, сколько памяти выделять на каждую подобную операцию! Следовательно, если у Вас 10 активных клиентов и каждый выполняет 1 сложный запрос, то значение в 10Мб для этого параметра скушает 100Мб оперативной памяти. Этот параметр стоит увеличивать, если у Вас большое количество памяти в распоряжении. Для старта следует выставить его в 1Мб.

wal_buffers

Этот параметр стоит увеличивать в системах с большим количеством записей. Значение в 1Мб рекомендуют разработчики postgres даже для очень больших систем.

synchronous_commit

Обратите особое внимание на этот параметр! Он *включает/выключает синхронную запись* в лог файлы после каждой транзакции. Это защищает от возможной потери данных. Но это накладывает ограничение на пропускную способность сервера.

Допустим, в Вашей системе не критична потенциально низкая возможность потери небольшого количества изменений при крахе системы. Но жизненно важно обеспечить в несколько раз большую производительность по количеству транзакций в секунду. В этом случае устанавливайте этот параметр в *off* (отключение синхронной записи).

Хранимые процедуры и функции

Хранимая процедура представляет собой подпрограмму, хранящуюся в базе данных. Она содержит имя, список параметров и операторы SQL. Все популярные системы управления базами данных поддерживают хранимые процедуры. Существует два вида подпрограмм: хранимые процедуры и функции, возвращающие значения, которые используются в других операторах SQL (например, pi()). Основное отличие заключается в том, что функции могут использоваться, как любое другое выражение в операторах SQL, а хранимые процедуры должны вызываться с помощью оператора CALL.

Они позволяют автоматизировать сложные процессы на уровне СУБД, нежели использовать для этого внешние скрипты. Это даёт нам наиболее высокую скорость выполнения, т.к. мы не гоняем большое количество запросов, а всего лишь один раз вызываем ту или иную процедуру (или функцию).

В чем преимущество хранимых процедур?

- Хранимые процедуры работают быстро. Преимущество сервера MySQL заключается в том, что он использует кэширование, а также заранее заданные операторы. Основной прирост скорости дает сокращение сетевого трафика. Если есть повторяющиеся задачи, которые требуют проверки, обработки циклов, нескольких операторов, и при этом не требуют взаимодействия с пользователем, это можно реализовать с помощью одного вызова процедуры, которая хранится на сервере;
- MySQL хранимые процедуры являются универсальными. При написании хранимой процедуры на SQL она будет работать на любой платформе, которая использует MySQL. В этом преимущество SQL над другими языками, такими как Java, C или PHP;
- Исходный код хранимых процедур всегда доступен в базе данных. Это эффективная практика связать данные с процессами, которые их обрабатывают.

В чем недостатки хранимых процедур?

- 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,PSQL,PL-SQL 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.

Создание хранимой процедуры

```
DELIMITER //
    CREATE PROCEDURE `p2` ()
    LANGUAGE SQL
```

```
DETERMINISTIC
SQL SECURITY DEFINER
COMMENT 'A procedure'
BEGIN
  SELECT 'Hello World !';
END //
```

Вызов хранимой процедуры

Чтобы вызвать хранимую процедуру, необходимо напечатать ключевое слово CALL, а затем название процедуры, а в скобках указать параметры (переменные или значения). Скобки обязательны

```
CALL stored_procedure_name (param1, param2, ....)
```

Параметры

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.

Простые переменные

```
DECLARE iVar INT DEFAULT 0;
 SET iVar = 5;
 SELECT * FROM `data` WHERE `id` = iVar;

 DECLARE iVar INT DEFAULT 0;
 SELECT COUNT(*) INTO iVar FROM `data`;
```

Системные переменные

```
SET @iVar = 5;
 SELECT @iVar;
```

Разница между простыми и системными переменными в том, что системные переменные доступны из вне хранимой процедуры. То есть, чтобы извлечь какие-то данные нужно пользоваться системными, а переменные которые нужны только внутри процедуры должны быть простыми.

Условия, Циклы. IF THEN ELSE, WHILE

Условия и циклы вам обязательно понадобятся при написании комплексных хранимых процедур, но зацикливаться на этой теме не буду. Думаю хоть какие-то навыки программирования у вас есть, так что покажу всего лишь синтаксис.

```
IF условие THEN
 действие;
 ELSE
```

```
действие;
END IF;

WHILE условие DO
действие;
END WHILE;
```

Курсоры

Курсоры используются для прохождения по набору строк, возвращенному запросом, а также обработки каждой строки

```
DECLARE cursor-name CURSOR FOR SELECT ...;      /*Объявление курсора и его заполнение */
    DECLARE  CONTINUE HANDLER FOR NOT FOUND        /*Что делать, когда больше нет записей*/
    OPEN cursor-name;                              /*Открыть курсор*/
    FETCH cursor-name INTO variable [, variable];   /*Назначить значение переменной, равной текущему значению столбца*/
    CLOSE cursor-name;
```

У курсоров есть три свойства, которые вам необходимо понять, чтобы избежать получения неожиданных результатов:

- Не чувствительный: открывшийся однажды курсор не будет отображать изменения в таблице, происшедшие позже. В действительности, MySQL не гарантирует то, что курсор обновится, так что не надейтесь на это.
- Доступен только для чтения: курсоры нельзя изменять.
- Без перемотки: курсор способен проходить только в одном направлении - вперед, вы не сможете пропускать строки, не выбирая их.

15

```
CREATE DEFINER=`root`@`%` FUNCTION `students1`() RETURNS int(11)
BEGIN

DECLARE student_id INT;
DECLARE exam_result INT;
DECLARE result INT;
declare id CHAR;

DECLARE done INT DEFAULT 0;

DECLARE scursor CURSOR FOR select student_id, result from student_result;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

OPEN scursor;

REPEAT
    FETCH scursor INTO student_id, exam_result;
    SET id = CONVERT(student_id, CHAR);

    IF id = "1" OR id="3" or id="5" or id="7" or id="9" THEN
       SET result = result + exam_result;
    ELSE
       SET result = result - exam_result;
    END IF;
UNTIL done END REPEAT;
CLOSE scursor;

RETURN result;
END
```

Триггеры

Триггер представляет собой именованный объект базы данных, который связан с таблицей, и он будет активизирован, когда специфическое событие происходит для таблицы.

```
CREATE TRIGGER trigger_name trigger_time trigger_event
ON tbl_name FOR EACH ROW trigger_stmt* This source code was highlighted with Source Code Highlighter.
```

trigger_name — название триггера
trigger_time — Время срабатывания триггера. BEFORE — перед событием. AFTER — после события.
trigger_event — Событие:
insert — событие возбуждается операторами insert, data load, replace
update — событие возбуждается оператором update
delete — событие возбуждается операторами delete, replace.
Операторы DROP TABLE и TRUNCATE не активируют выполнение триггера

tbl_name — название таблицы
trigger_stmt выражение, которое выполняется при активации триггера

```
CREATE TRIGGER `update_test` AFTER INSERT ON `test`
FOR EACH ROW BEGIN
  INSERT INTO log Set msg = 'insert', row_id = NEW.id;
END;
```