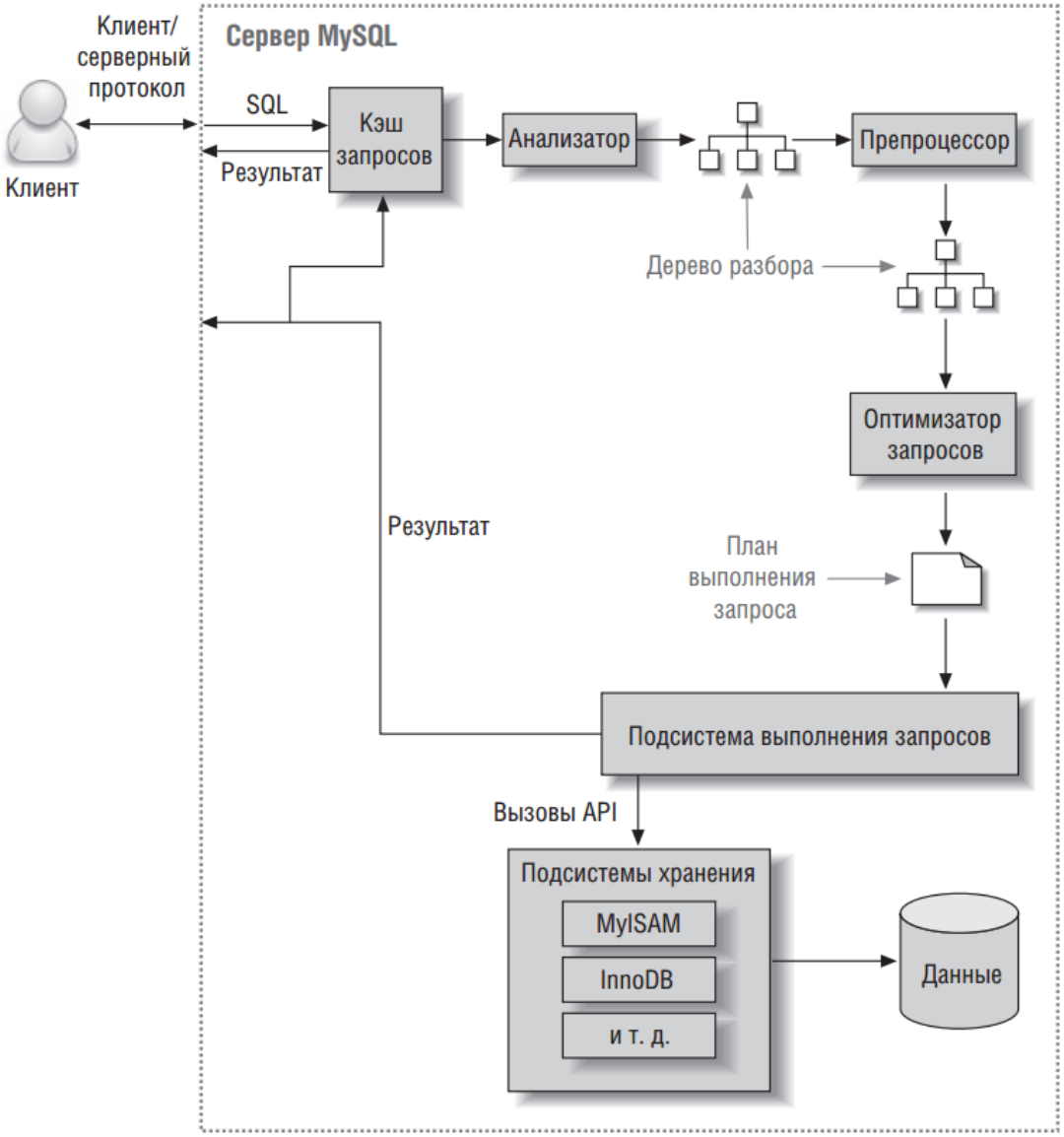
10. Оптимизация запросов

к.т.н., доцент кафедры ИиСП

Лучинин
Захар Сергеевич

Логическая архитектура сервера MySQL



Оптимизация запроса

- **Изменение порядка соединения**

  Таблицы не обязательно соединять именно в том порядке, который указан в запросе.

- **Применение алгебраических правил эквивалентности**

  (5=5 AND a>5) **->** a>5

  (a < b AND b=c) AND a=5 **->** b>5 AND b=c AND a=5

- **Оптимизации COUNT(), MIN() и MAX()**

  Наличие индексов и сведений о возможности хранения NULL-значений в столбцах часто позволяет вообще не вычислять эти выражения.

Оптимизация запроса

- **Покрывающие индексы**

  Если индекс содержит все необходимые запросу столбцы, то СУБД может воспользоваться им, вообще не читая данные таблицы.

- **Оптимизация подзапросов**

  СУБД умеет преобразовывать некоторые виды подзапросов в более эффективные эквивалентные формы, сводя их к поиску по индексу.

- **Раннее завершение**

  СУБД может прекратить обработку запроса (или какой-то шаг обработки), как только поймет, что этот запрос или шаг полностью выполнен.

План выполнения

- Множество способов выполнить запрос
- Нужно выбрать лучший
- «Стоимость» запроса
  - количество страниц в таблице или в индексе
  - кардинальность индекса
  - длина строк и ключей
  - распределение ключей в индексе

Почему оптимизатор ошибается

- Некорректная статистика
- Быстрее – не значит дешевле
- Не учитывается параллельно выполняющиеся запросы
- Не учитывается стоимость выполнения хранимых процедур и UDF
- …

Как повлиять на оптимизатор

- Переписать запрос
- Создавать индексы
- Использовать подсказки в запросах (Hints)
- Изменить настройка СУБД
- Обслуживание СУБД (обновлять статистику)

Тестовая база

Таблица	Кортежей	Размер
genres	19	48 kB
links	40 110	2 944 kB
movie_genres	74 229	4 856 kB
movie_tags	668 953	49 152 kB
movies	40 110	3 400 kB
rating	24 404 096	1 971 200 kB

Общий размер: ~1995MB
Запрос для примера: вывести все комедии про зомби по рейтингу.

Рейтинг комедий про зомби

- **Начальный вариант (7.7 secs):**

```
SELECT m.id, m.title, avg(r.rating)
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)
JOIN ratings r ON (r.movie_id = m.id)
WHERE lower(g.name) = lower('Comedy')
  AND lower(t.name) LIKE lower('Zombie%')
GROUP BY m.id,
         m.title
ORDER BY avg(r.rating) DESC;
```

Рейтинг комедий про зомби

- **Построили типовые индексы (10.9 secs):**

```
SELECT m.id, m.title, avg(r.rating)
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)
JOIN ratings r ON (r.movie_id = m.id)
WHERE lower(g.name) = lower('Comedy')
  AND lower(t.name) LIKE lower('Zombie%')
GROUP BY m.id,
         m.title
ORDER BY avg(r.rating) DESC;
```

Рейтинг комедий про зомби

- **Немного переписали запрос (486 msec):**

```
SELECT m.id, m.title, avg(r.rating)
FROM
  (SELECT m.id,
          m.title
   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%')
   GROUP BY m.id) m
JOIN ratings r ON (r.movie_id = m.id)
GROUP BY m.id,
         m.title
ORDER BY avg(r.rating) DESC;
```

Рейтинг комедий про зомби

- **Покрывающий индекс по рейтингам (82 msec):**

```
SELECT m.id, m.title, avg(r.rating)
FROM
  (SELECT m.id,
          m.title
   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%')
   GROUP BY m.id) m
JOIN ratings r ON (r.movie_id = m.id)
GROUP BY m.id,
         m.title
ORDER BY avg(r.rating) DESC;
```

Оптимизируйте доступ к данным

- Не извлекает ли приложение больше данных, чем нужно;
- Не анализирует ли сервер больше строк, чем это необходимо;

Типичные ошибки:

- Выборка ненужных строк;
- Выборка всех столбцов из соединения нескольких таблиц;
- Выборка всех столбцов.

SUBQUERIES vs JOIN

```sql
-- 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);
```

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.

Когда:

- Большое количество соединений таблиц
- Большое количество расчетов

При денормализации данных вырастет количество записей, но уменьшится количество чтений.

Пример денормализации

До

reader

	Column Name	Condensed Type	Nullable
🔑	id_reader	int	No
	first_name	nvarchar(50)	No
	last_name	nvarchar(100)	No
	reader_num	nchar(100)	No

issuance

	Column Name	Condensed Type	Nullable
🔑	id_issuance	int	No
	id_copy	int	No
	id_reader	int	No
	issue_date	date	No
	release_date	date	No
	deadline_date	date	No

После

reader

	Column Name	Condensed Type	Nullable
🔑	id_reader	int	No
	first_name	nvarchar(50)	No
	last_name	nvarchar(100)	No
	reader_num	nchar(100)	No
	book_count	int	No

Подходы при денормализации

- **Дублирование**.
  + Избавляемся от JOIN’ов
  - Обновлять данные в нескольких таблицах

- **Предварительная подготовка**.
  + Не тратим время на расчеты во время запроса
  - Обновлять данные в одной и более таблицах

- **Группировка данных**.
  + Хранение нескольких значений в одном атрибуте
  - Нет возможности осуществлять поиск

Как реализовать денормализацию

- Сохранить детальные таблицы
- Использовать триггеры
- Программная поддержка

Оптимизация DELETE

Очистка таблицы

```
DELETE FROM film;
```

Для удаления всех записей из таблицы/таблиц есть отдельная команда

```
TRUNCATE TABLE film;
```

Особенности:

TRUNCATE на много быстрее, чем DELETE;

TRUNCATE нарушает изоляцию транзакций.

Слияние индексов

```sql
-- Использование индекса только по одному атрибуту
SELECT film_id, actor_id FROM film_actor
WHERE actor_id = 1 OR film_id = 1;

-- Использование индекса по двум атрибутам
SELECT film_id, actor_id FROM film_actor
WHERE actor_id = 1
UNION ALL
SELECT film_id, actor_id FROM film_actor
WHERE film_id = 1 AND actor_id <> 1;
```

Оптимизация GROUP BY и DISTINCT

```sql
-- До оптимизации
SELECT actor.first_name, actor.last_name, COUNT(*)
FROM film_actor
INNER JOIN actor USING(actor_id)
GROUP BY actor.first_name, actor.last_name;

-- После оптимизации
SELECT actor.first_name, actor.last_name, c.cnt
FROM actor
INNER JOIN (
    SELECT actor_id, COUNT(*) AS cnt
    FROM film_actor
    GROUP BY actor_id
) AS c USING(actor_id);
```

Оптимизация LIMIT со смещением

```sql
-- До оптимизации
SELECT film_id, description
FROM film
ORDER BY title
LIMIT 50, 5;

-- После оптимизации
SELECT film.film_id, film.description
FROM film
INNER JOIN (
-- Использование индекса для поиска
    SELECT film_id FROM film
    ORDER BY title LIMIT 50, 5
) AS lim USING(film_id);
```

Оптимизация COUNT(*)

- **Получение кол-ва записей после выполнения запроса**

```
SELECT COUNT (*) FROM tags WHERE name LIKE 'Zombie%';
```

Если нужно ориентировочное количество записей в результате выполнения запроса, то можно получить их из плана выполнения:

```
EXPLAIN SELECT * FROM tags WHERE name LIKE 'Comedy%';
```

Оптимизация переменных

- Не надо генерировать повторно план выполнения
- https://habr.com/ru/post/573438/

Оптимизация на уровне приложения

Кэш запросов

Обращение к БД дороже, чем обращение к файловой системе. Прочитав значение из БД, сохраняем результат, например, в файловый кэш.

При изменении записи необходимо инвалидировать кэш.

Декомпозиция соединения

```sql
-- Найти все новости с тегом sql одним запросом
SELECT * FROM tag
    JOIN news_tag ON news_tag.id_tag = tag.id_tag
    JOIN news ON news_tag.id_news = news.id_news
WHERE tag.name = 'sql';
```

```sql
-- Найти все новости с тегом sql тремя запросами
SELECT * FROM tag WHERE tag = 'sql';
SELECT * FROM news_tag WHERE id_tag = 1234;
SELECT * FROM news WHERE id_news IN (123,456,567,9098,8904);
```

Декомпозиция соединения

Соединение в приложении может оказаться эффективнее в следующих случаях:

- Организован кэш и вы повторно используете ранее запрошенные данные
- Данные распределены по нескольким серверам
- Вместо соединения с большой таблицей используется список IN()

Пакетная обработка

Пакетная обработка запросов может быть эффективнее из-за:

- Возможности избежать задержки в сети при обращении к базе данных SQL;
- Архитектуры базы данных;
- Использования несколько баз данных (сегментирование/шардинг).

https://docs.microsoft.com/ru-ru/azure/sql-database/sql-database-use-batching-to-improve-performance

Вставка данных в транзакции

```
SqlTransaction transaction = conn.BeginTransaction();

foreach (string commandString in dbOperations)
{
   SqlCommand cmd = new SqlCommand(commandString, conn, transaction);
   cmd.ExecuteNonQuery();
}

transaction.Commit();
```

Операции вставки	Без транзакций (мс)	С транзакциями (мс)
1	21	26
10	220	56
100	2145	341
1000	21 479	2756

Многострочные инструкции INSERT

```
-- Вставка 3 запросами
INSERT INTO artist (name) VALUES ('Bud Powell');

INSERT INTO artist (name) VALUES ('Candido');

INSERT INTO artist (name) VALUES ('Charlie Byrd');

-- Вставка одним запросом. Предпочтительный вариант
INSERT INTO artist (name)
VALUES ('Buddy Rich'),
       ('Candido'),
       ('Charlie Byrd');
```

Группировка UPDATE

```
-- Решение в лоб. Выполнение всех операций построчно
UPDATE items SET res_id = 73534, level = 1, meta = 1001
WHERE res_id = 40477;
UPDATE items SET res_id = 73534, level = 1, meta = 1201
WHERE res_id = 40478;
UPDATE items SET res_id = 73534, level = 2, meta = 1031
WHERE res_id = 40479;
…
UPDATE items SET res_id = 73534, level = 80, meta = 7641
WHERE res_id = 70477;
```

Группировка UPDATE

```
-- создаем временную таблицу
CREATE TEMPORARY TABLE tmp_items (
  res_id INT,
  level  SMALLINT,
  meta   BYTEA
);

-- вставляем новые значения во временную таблицы
INSERT INTO items VALUES
  (1, 1001, 40477),
  (1, 1201, 40478),
  …
  (80, 7641, 70477);

-- обновляем
UPDATE items i
SET res_id = 73534, level = t.level, meta = t.meta
FROM tmp_items t
WHERE i.res_id = t.res_id;
```

См. также оператор MERGE

Параллельная обработка

Попытка вставить 1000 строк в одну или несколько параллельных пакетов. Этот тест показывает, как большее количество одновременных пакетов фактически снижает производительность.

Размер пакета [количество итераций]	Два потока (мс)	Четыре потока (мс)	Шесть потоков (мс)
1000 [1]	277	315	266
500 [2]	548	278	**256**
250 [4]	405	329	265
100 [10]	488	439	391

После достижения неопределенного порогового значения большее число потоков уменьшит производительность, а не повысит ее.

Работа с транзакцией

Уменьшение времени блокировок за счет избавления от крупных и сложных запросов

- Разбиение запроса на более мелкие;
- Сокращайте время транзакции;
- Направленное изменение записей с целью устранение возникновения deadlock;
- Оптимистичные блокировки.

Что такое ORM

Объектно-Реляционное Отображение (*Object-Relational Mapping*) -технология, связывающая модель базы данных и концепции ООП.

Позволяет работать с данными как с объектами.

Преимущества ORM

- Представление модели данных в ORM независимо от СУБД
- Упрощенное моделирование базы данных
- ORM предоставляет больше механизмов обеспечения целостности данных
- Возможность использовать наследование моделей

Недостатки ORM

- Медленее, чем “голый” SQL
- Возможны проблемы со “сложными” запросами
- Идея всех ORM схожа, но реализации различны: нужно дополнительное обучение