Nota de aula - Otimização de consultas no MySQL

Baseada em:
Schwartz, B.; Zaitsev, P.; Tkachenko, V.; D. Zawodny, J.; Lentz, A.; Balling, D. J.
High Performance MySQL. O'REILLY: 2008.

1 OTIMIZAÇÃO DE ESQUEMA E INDEXAÇÃO

"Se você quer alta performance, você deve projetar seu banco de dados e seus índices para as consultas específicas que você irá executar". Otimização geralmente envolve compensação. Por exemplo, adicionar índices para acelerar o retorno dos dados irá atrasar as atualizações de registros. Semelhantemente, um banco de dados não-normalizado pode acelerar alguns tipos de consultas mas atrasar outras. Adicionar tabelas para contar ou somar é uma ótima maneira para otimizar consultas, mas pode ser custoso para manter.

1.1 Escolhendo tipos de dados ótimos

MySQL suporta uma grande variedade de tipos de dados, escolher o tipo correto para armazenar seus dados é crucial para ter uma boa performance.

Algumas diretrizes:

- **O menor é geralmente melhor:** em geral, tente usar o menor tipo de dado que pode armazenar e representar seus dados. Tipos de dados menores são geralmente mais rápidos, porque eles usam menos espaço em disco, em memória e no cache da CPU. Eles ainda requerem poucos ciclos de CPU para serem processados.
- **Simples é bom:** por exemplo, inteiros são mais baratos (computacionalmente) para comparar do que caracteres, porque conjuntos de caracteres e *collations* fazem comparações de caracteres complicadas. Armazene data e hora no formato correto em MySQL e não como string.
- **Evite NULL se possível:** você deveria definir campos como NOT NULL sempre que puder. É difícil para o MySQL otimizar consultas que se referem a colunas que podem ser nulas, porque elas fazem os índices, as estatísticas de índices e os valores de comparação mais complicados. Uma coluna nula usa mais espaço de armazenamento e requer um processamento especial dentro do MySQL.
- **Generosidade pode ser imprudente:** armazenar o valor "hello" requer a mesma quantidade de espaço em um VARCHAR(5) e em um VARCHAR(200). Porém, há uma grande vantagem em especificar menor caracteres. A coluna maior pode usar mais memória, porque o MySQL frequentemente aloca blocos de memória de tamanho fixo para manter os valores internamente. Isso é especialmente ruim para ordenação ou operação que usam tabelas temporárias em memória. A mesma coisa acontece com ordenações de arquivos que usam tabelas temporárias em disco. A melhor estratégia é alocar quanto espaço for necessário somente.
- **BLOB e TEXT:** MySQL ordena BLOB e TEXT diferentemente dos outros tipos: ao invés de ordenar o comprimento total da string, ele ordena apenas os primeiros MAX_SORT_LENGTH bytes da coluna. Então, é possível diminuir o tamanho dessa variável de servidor ou usar ORDER BY SUBSTRING(column, length).
- **Tipos de dados de chaves primárias:** Inteiros são a melhor escolha, porque são mais rápidos e trabalham com AUTO_INCREMENT. Strings devem ser evitadas como chave primárias.

1.2 Indexação

Índices são estruturas de dados que ajudam o MySQL a recuperar dados eficientemente. Eles são críticos para uma boa performance. São também chamados de "chaves" (*keys*) em MySQL.

A maneira mais fácil para entender como um índice funciona em MySQL é pensar em um índice de um livro. Para encontrar um tópico específico discutido no livro, você olha o índice e vai até a página em que o tópico aparece. O MySQL usa índices de uma maneira semelhante. Ele busca um valor na estrutura de dados de índice. Quando o encontra, ele pode encontrar a linha que contém o valor procurado. Suponha a seguinte consulta:

```
mysql> SELECT first_name FROM sakila.actor WHERE actor_id = 5;
```

Se houver um índice na coluna **actor_id**, então o MySQL irá usar o índice para encontrar as linhas em que a coluna **actor_id** é igual a 5. Em outras palavras, ele buscará os valores no índice e retornará as linhas correspondentes.

Um índice contém valores de uma ou várias colunas de uma tabela. Se um índice tem mais de uma coluna, a ordem das colunas é importante, porque o MySQL por apenas buscar de forma eficiente na coluna mais à esquerda do índice. Criar um índice de duas colunas não é o mesmo que criar dois índices de uma coluna.

Estratégias de indexação para alta performance

Há muitas maneiras de escolher e usar os índice efetivamente, porque há muitos casos específicos de otimização.

Algumas estratégias:

Isole a coluna
se você não isolar a coluna indexada em uma consulta, o MySQL geralmente não poderá usar os índices da coluna. Isolar a coluna significa que ela não deveria ser parte de uma expressão ou estar dentro de uma função na consulta.
Por exemplo, esta é uma consulta que não usará o índice na coluna **actor_id**:

```
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;
```

A coluna **actor_id** deveria estar isolada, ou seja, sem a operação de soma.

Outro exemplo em que o índice não seria usado:
```
mysql> SELECT … WHERE TO_DAYS(CURRENT_DATE) - TO_DAYS(date_col) <= 10;
```

Uma solução melhor é:
```
mysql> SELECT … WHERE date_col >= DATE_SUB(CURRENT_DATE, INTERVAL 10 DAY);
```

Índices de prefixo e seletividade de índices
Algumas vezes você precisa indexar colunas com muitos caracteres, o que pode tornar o índice grande e lento. Uma estratégia é simular um *hash index*. Mas isso pode não ser suficiente. O que fazer?

Você pode economizar espaço e ter uma boa performance indexado os primeiros caracteres da string. Isso faz seu índice usar menos espaço, mas pode fazê-lo menos **seletivo**. O índice de

seletividade é a taxa do número de valores indexados distintos (cardinalidade) em relação ao total de linhas na tabela. Um índice UNIQUE tem seletividade 1.

Para determinar um bom comprimento de prefixo, encontre os valores mais frequentes e compare a lista com uma lista dos prefixos mais frequentes.

Exemplo: conta a quantidade de registros agrupando pelos 3 caracteres iniciais da coluna. Quanto mais valores únicos, melhor. Então, talvez valha a pena aumentar o tamanho do prefixo neste caso.

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 3) AS pref
-> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 10;
+-----+------+
| cnt | pref |
+-----+------+
| 483 | San |
| 195 | Cha |
| 177 | Tan |
| 167 | Sou |
| 163 | al- |
| 163 | Sal |
| 146 | Shi |
| 136 | Hal |
| 130 | Val |
| 129 | Bat |
+-----+------+
```

Outra maneira de avaliar é calcular o índice de seletividade como explicado anteriormente.

```
mysql> SELECT COUNT(DISTINCT LEFT(city, 3))/COUNT(*) AS sel3,
-> COUNT(DISTINCT LEFT(city, 4))/COUNT(*) AS sel4,
-> COUNT(DISTINCT LEFT(city, 5))/COUNT(*) AS sel5,
-> COUNT(DISTINCT LEFT(city, 6))/COUNT(*) AS sel6,
-> COUNT(DISTINCT LEFT(city, 7))/COUNT(*) AS sel7
-> FROM sakila.city_demo;
+--------+--------+--------+--------+--------+
| sel3 | sel4 | sel5 | sel6 | sel7 |
+--------+--------+--------+--------+--------+
| 0.0239 | 0.0293 | 0.0305 | 0.0309 | 0.0310 |
+--------+--------+--------+--------+--------+
```

Nesse caso, o índice com 7 caracteres de prefixo é mais adequado.

Cobertura de índices

Um índice que contém (ou "cobre") todos os dados necessários para satisfazer uma consulta é chamado **índice de cobertura**, que podem ser uma ferramenta poderosa e podem melhorar a performance significativamente. Considere os seguintes benefícios de ler os dados somente através de índices:

- Valores dos índices são geralmente menores que o tamanho total da linha, então o MySQL pode acessar significativamente menos dados se ele ler o índice.
- Índices são armazenados por seus valores de índice (ao menos dentro da página), então há menos operação de I/O em relação a acessar a linha completa dos dados.
- Muitos mecanismos de armazenamento usam cache de índices melhor que de dados.

No resultado do comando EXPLAIN, a expressão "Using index" na coluna Extra indica que a consulta é coberta por um índice.
Observação: o MySQL não executa a operação LIKE em índices, é uma limitação.

Usando varredura de índices para ordenação

MySQL tem duas maneiras para produzir resultados ordenados: usar uma ordenação de arquivo ou varrer um índice em ordem. No comando EXPLAIN, quando o MySQL faz uma varredura de índice (index scan), a palavra "index" é exibida na coluna **type**.
Varrer o índice é mais rápido, porque simplesmente requer mover de uma entrada do índice para outra. Entretanto, se o MySQL não usar o índice para cobrir a consulta, ele terá que verificar se cada linha existe no índice.

MySQL pode usar o mesmo índice para ordenar ou filtrar linhas. Se possível, tente projetar seus índices para ambas as tarefas.

Exemplo: considere a seguinte tabela.

```
CREATE TABLE rental (
...
PRIMARY KEY (rental_id),
UNIQUE KEY rental_date (rental_date,inventory_id,customer_id),
KEY idx_fk_inventory_id (inventory_id),
KEY idx_fk_customer_id (customer_id),
KEY idx_fk_staff_id (staff_id),
...
);
```

O MySQL usa o índice "rental_date" para ordenar a seguinte consulta:

```
mysql> EXPLAIN SELECT rental_id, staff_id FROM sakila.rental
-> WHERE rental_date = '2005-05-25'
-> ORDER BY inventory_id, customer_id\G
*************************** 1. row ***************************
type: ref
possible_keys: rental_date
key: rental_date
rows: 1
Extra: Using where
```

Isso acontece porque foi especifica uma condição de igualdade para a primeira coluna do índice.

Alguns exemplos em que o índice não será usado:

- Esta consulta usa duas direções de ordenação, mas as colunas do índices estão ordenadas crescentemente.

  ```
  ... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC, customer_id ASC;
  ```

- Aqui, o ORDER BY se refere a uma coluna que não está no índice.

  ```
  ... WHERE rental_date = '2005-05-25' ORDER BY inventory_id, staff_id;
  ```

- Aqui, o WHERE e o ORDER BY não considerar o prefixo mais à esquerda do índice.

  ```
  ... WHERE rental_date = '2005-05-25' ORDER BY customer_id;
  ```

- Esta consulta tem uma faixa de condição da primeira coluna, então o MySQL não usa o resto do índice.

  ```
  ... WHERE rental_date > '2005-05-25' ORDER BY inventory_id, customer_id;
  ```

- Aqui há uma igualdade múltipla na coluna "inventory_id". Para os propósitos de ordenação, isso é mesmo que uma faixa.

  ```
  ... WHERE rental_date = '2005-05-25' AND inventory_id IN(1,2) ORDER BY customer_
  ```

```
id;
```

Índices redundantes e duplicados
MySQL permite criar múltiplos índices para uma mesma coluna. Índices duplicados são índices do mesmo tipo, criados no mesmo conjunto de colunas na mesma ordem. Evite criar índices duplicados.

Índices redundantes são diferentes de índices duplicados. Se há um índice em (A,B), outro índice em (A) seria redundante porque é um prefixo do primeiro índice. Isto é, o índice em (A,B) pode também ser usado como um índice em (A). Entretanto, um índice em (B,A) não seria redundante e nem um índice em (B), porque B não é um prefixo do índice (A,B).

Exemplo: há um índice em "state_id", que é útil para a seguinte consulta:

```
mysql> SELECT count(*) FROM userinfo WHERE state_id=5;
```

Há ainda uma consulta relacionada que recupera várias colunas ao invés de somente contar linhas.

```
mysql> SELECT state_id, city, address FROM userinfo WHERE state_id=5;
```

Para melhorar essa consulta, a solução é estender o índice para (state_id, city, address), então o índice cobrirá a consulta.

Evite múltiplas condições de faixa
Observe a seguinte consulta:

```
WHERE eye_color IN('brown','blue','hazel')
AND hair_color IN('black','red','blonde','brown')
AND sex IN('M','F')
AND last_online > DATE_SUB('2008-01-17', INTERVAL 7 DAY)
AND age BETWEEN 18 AND 25
```

Há um problema com essa consulta: ela tem duas condições de faixa. O MySQL pode usar o critério "last_online" ou "age", mas não ambos.

Otimização ordenações
É possível adicionar índices especiais para os casos de baixa seletividade. Por exemplo, um índice em (sex, rating) pode ser usado para a seguinte consulta:

```
mysql> SELECT <cols> FROM profiles WHERE sex='M' ORDER BY rating LIMIT 10;
```

Analisando a cardinalidade dos índices
É possível verificar a cardinalidade dos índices através do comando SHOW INDEX FROM. Exemplo:

```
mysql> SHOW INDEX FROM sakila.actor\G
*************************** 1. row ***************************
Table: actor
Non_unique: 0
Key_name: PRIMARY
Seq_in_index: 1
Column_name: actor_id
Collation: A
Cardinality: 200
Sub_part: NULL
Packed: NULL
Null:
```

```
Index_type: BTREE
Comment:
*************************** 2. row ***************************
Table: actor
Non_unique: 1
Key_name: idx_actor_last_name
Seq_in_index: 1
Column_name: last_name
Collation: A
Cardinality: 200
Sub_part: NULL
Packed: NULL
Null:
Index_type: BTREE
Comment:
```

A informação de cardinalidade indica quantos valores distintos o mecanismo de armazenamento estima existirem no índice.

1.4 Normalização e Denormalização

Prós e contras de um banco de dados normalizado

- Updates normalizados são geralmente mais rápidos que updates não-normalizados.
- Quando um data está bem normalizado, há pouco ou nenhum dado duplicado, então há menos dado para mudar.
- Tabelas normalizadas geralmente são menores, então se ajustam na memória e executam melhor.

As desvantagens de bancos de dados normalizados estão relacionadas ao uso de JOIN para selecionar informações. Isso pode prejudicar o uso de estratégias de índices.

Prós e contras de um banco de dados não-normalizado

- Evitar junções de tabelas.
- Facilita o uso de índices, devido aos dados relacionados estarem em uma mesma tabela muitas vezes.

Um mistura de normalizado e não-normalizado

A maneira mais comum de denormalizar dados é duplicá-los, usar cache ou colunas selecionadas de uma tabela em outra. Em MySQL, é possível usar triggers para atualizar os valores duplicados.

2 Otimização de performance de consulta

Algumas consultas solicitam mais dados do que necessitam. Isso demanda trabalho extra do servidor MySQL, sobrecarrega a rede e consome memória e CPU.

Alguns erros típicos:

- **Retornar mais linhas do que o necessário:** a melhor solução para evitar isso é usar a cláusula LIMIT.
- **Selecionar todas as colunas de um JOIN com múltiplas tabelas:** se você quer recuperar todos os atores de um filme, não escreva esta consulta:

```
mysql> SELECT * FROM sakila.actor
-> INNER JOIN sakila.film_actor USING(actor_id)
-> INNER JOIN sakila.film USING(film_id)
        -> WHERE sakila.film.title = 'Academy Dinosaur';
```

Execute esta:

```
mysql> SELECT sakila.actor.* FROM sakila.actor...;
```

- **Selecionar todas as colunas:** o SELECT * FROM pode impedir o uso de índices de cobertura.

Alguns tipos de otimização de consultas MySQL:

- **Reordenação de joins:** tabelas nem sempre são recuperadas na ordem que você especifica na consulta. Determinar a melhor ordem de JOIN é uma importante otimização.
- **Converter OUTER JOIN para INNER JOIN:** se um OUTER JOIN estiver sendo executado na consulta sem retornar resultados diferentes de um INNER JOIN, então substitua-o.
- **Aplicar regras algébricas equivalentes:** MySQL aplica transformações algébricas para simplificar expressões. Por exemplo, o termo (5=5 AND a > 5) pode ser reduzido para (a > 5). Semelhantemente, (a < b AND b = c) AND a = 5 pode ser reduzido para (b > 5 AND b=c AND a = 5).
- **Otimização de COUNT(), MIN(), MAX():** o uso de índices para encontrar valores mínimos e máximos otimizar o retorno das consultas, porque, para um valor mínimo, basta selecionar a primeira linha do índice.
- **Otimização de subconsulta:** elimine as subconsultas quando possível, substitua por INNER JOIN ou até mesmo faça consultas separadas.
- **Propagação de igualdade:** MySQL reconhece quando uma consulta tem duas colunas iguais (em um JOIN, por exemplo) e propaga a cláusula WHERE entre as colunas equivalentes. Exemplo:

```
mysql> SELECT film.film_id
-> FROM sakila.film
-> INNER JOIN sakila.film_actor USING(film_id)
        -> WHERE film.film_id > 500;
```

Na consulta acima, o MySQL reconhece que o WHERE deve ser propagado para o campo film_id das tabelas sakila.film e sakila.film_actor. Em outros SGBD, seria necessário especificar a condição:

```
... WHERE film.film_id > 500 AND film_actor.film_id > 500
```

- **Otimização de JOIN:** certifique-se de há índices nas colunas das cláusulas ON ou USING. Tente assegurar que qualquer GROUP BY ou ORDER BY se refira a coluna de apenas uma tabela, então o MySQL pode tentar usar um índice para a operação.
- **Otimizando GROUP BY e DISTINCT:** é mais eficiente agrupar pela chave primária do que por outros campos.

Dicas para o otimizador de consultas
O MySQL permite indicar dicas explicitamente ao otimizador de consultas. As opções são:

- **HIGH_PRIORITY e LOW_PRIORITY:** indicam como priorizar o comando em relação a outros comandos que estão tentando acessar as mesmas tabelas.
- **DELAYED:** esta opção é usada com INSERT e UPDATE. Atrasa a inserção ou atualização para o momento em que a tabela estiver livre. É mais útil para aplicações de log ou similares.
- **STRAIGHT_JOIN:** pode ser usada depois da palavra SELECT ou em qualquer comando entre duas tabelas com JOIN. O primeiro uso força todas as tabelas na consulta a serem relacionadas na ordem em que elas estão listadas no comando. O segundo uso força a ordem de JOIN nas duas tabelas em que o STRAIGHT_JOIN aparece.
- **SQL_SMALL_RESULT e SQL_BIG_RESULT:** em comandos SELECT, estas opções indicam ao otimizador quando e como usar as tabelas temporárias. SQL_SMALL_RESULT informa o otimizador de que o resultado será pequeno e que pode ser colocado em tabelas temporárias indexadas para evitar ordenação e agrupamento; SQL_BIG_RESULT indica que o resultado será grande e que é melhor usar tabelas temporárias no disco com ordenação.
- **SQL_CACHE e SQL_NO_CACHE:** indica ao otimizador que a consulta é ou não candidata para o cache de consultas.
- **USE INDEX, IGNORE INDEX e FORCE INDEX:** indicam ao otimizador para usar ou ignorar índices. FORCE INDEX é o mesmo que USE INDEX, mas diz ao otimizador que a varredura da tabela é extremamente custosa se comparada ao uso do índice, mesmo que o índice não seja útil.

3 Usando o EXPLAIN

O comando EXPLAIN é a principal maneira de saber como o otimizador de consultas realiza suas decisões.
Para usar o EXPLAIN, basta adicioná-lo antes da consulta SELECT.

Exemplo: **EXPLAIN SELECT * FROM USUARIOS**

EXPLAIN EXTENDED é exibido como o EXPLAIN, mas diz ao servidor para fazer uma compilação reversa do plano de execução do SELECT.
EXPLAIN PARTITIONS mostra as partições que a consulta irá acessar, se aplicável.

Colunas do EXPLAIN

- **id:** sempre contém um número que identifica a que linha o SELECT pertence. Se não há subconsultas ou UNION no comando, há apenas uma linha SELECT.

```
mysql> EXPLAIN SELECT (SELECT 1 FROM sakila.actor LIMIT 1) FROM sakila.film;
+----+-------------+-------+...
| id | select_type | table |...
+----+-------------+-------+...
| 1 | PRIMARY | film |...
| 2 | SUBQUERY | actor |...
          +----+-------------+-------+...
```

```
mysql> EXPLAIN SELECT film_id FROM (SELECT film_id FROM sakila.film) AS der;
+----+-------------+------------+...
| id | select_type | table |...
+----+-------------+------------+...
| 1 | PRIMARY | <derived2> |...
| 2 | DERIVED | film |...
          +----+-------------+------------+...
```

- **select_type:** mostra se é um SELECT simples ou complexo. O valor SIMPLE significa que a consulta não contém subconsultas ou UNION. Se a consulta tem qualquer subparte complexa, a parte é rotulada com PRIMARY e outras partes são rotuladas como segue:
    - SUBQUERY: há uma subconsulta.
    - DERIVED: usado para um SELECT que está contido em uma subconsulta na cláusula FROM.
    - UNION: o segundo e os demais SELECT são rotulados como UNION.
    - UNION RESULT: o SELECT usado para retornar os resultados de uma tabela temporária de UNION é rotulado como UNION RESULT.
- **table:** mostra qual tabela a linha do EXPLAIN está acessando.

```
mysql> EXPLAIN SELECT film.film_id
-> FROM sakila.film
-> INNER JOIN sakila.film_actor USING(film_id)
-> INNER JOIN sakila.actor USING(actor_id);
          +----+-------------+------------+...
| id | select_type | table |...
+----+-------------+------------+...
| 1 | SIMPLE | actor |...
| 1 | SIMPLE | film_actor |...
| 1 | SIMPLE | film |...
          +----+-------------+------------+...
```

- **type:** mostra o tipo de JOIN, indicando o tipo de acesso. As opções são:
    - ALL: a tabela é varrida do começo ao fim para encontrar a linha.

    - INDEX: é o mesmo que ALL, exceto que a varredura é no índice. A principal vantagem é evitar ordenação; a maior desvantagem é o custo de ler a tabela inteira na ordem do índice.
    - RANGE: é um INDEX SCAN limitado. Inicia em algum ponto do índice e retorna linha que encontram uma faixa de valores. É melhor que um FULL INDEX SCAN. Range scans são resultantes de BETWEEN ou > na cláusula WHERE.
    - REF: é um acesso ao índice que retorna um valor único. É chamado de **ref** porque é comparado a um valor de referência.
    - EQ_REF: é uma varredura do índice que retorna mais de um valor.
    - NULL: o otimizador não conseguiu resolver o tipo de acesso.
- **possible_keys:** mostra quais índices poderiam ser usado para a consulta, baseado nas consultas e nos operadores usados.
- **key:** mostra qual índice dos **possible_keys** o otimizador decidiu usar para minimizar o custo da consulta.

```
mysql> EXPLAIN SELECT actor_id, film_id FROM sakila.film_actor\G
*************************** 1. row ***************************
id: 1
select_type: SIMPLE
table: film_actor
type: index
possible_keys: NULL
key: idx_fk_film_id
key_len: 2
ref: NULL
rows: 5143
        Extra: Using index
```

- **key_len:** mostra o número de bytes que o MySQL irá usar no índice. Lembrando: o MySQL usa apenas o prefixo mais à esquerda.
- **ref:** mostra quais colunas ou constantes das tabelas precedentes estão sendo usadas para procurar valores no índice nomeado na coluna **key.**
- **rows:** número de linhas, em média, que o MySQL pensa que terá que ler para encontrar as linhas que satisfazem a consulta.
- **filtered:** é uma estimativa pessimista da porcentagem de linhas que satisfará alguma condição da tabela, tais como WHERE ou JOIN.
- **extra:** as opções são:
    - USING INDEX: indica que o MySQL usará um índice de cobertura para evitar acessar a tabela.
    - USING WHERE: significa que as linhas serão filtradas após serem recuperadas.
    - USING TEMPORARY: significa que será usada uma tabela temporária enquanto o resultado é ordenado.
    - USING FILESORT: o MySQL usará uma ordenação externa para ordenar os resultados, ao invés de ler a tabela na ordem do índice.
    - RANGE CHECKED FOR EACH RECORD: não há um bom índice.